

AD-A066 386

MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/2
EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS FOR DATABASE SYS--ETC(U)
JAN 79 S A BORKIN

UNCLASSIFIED

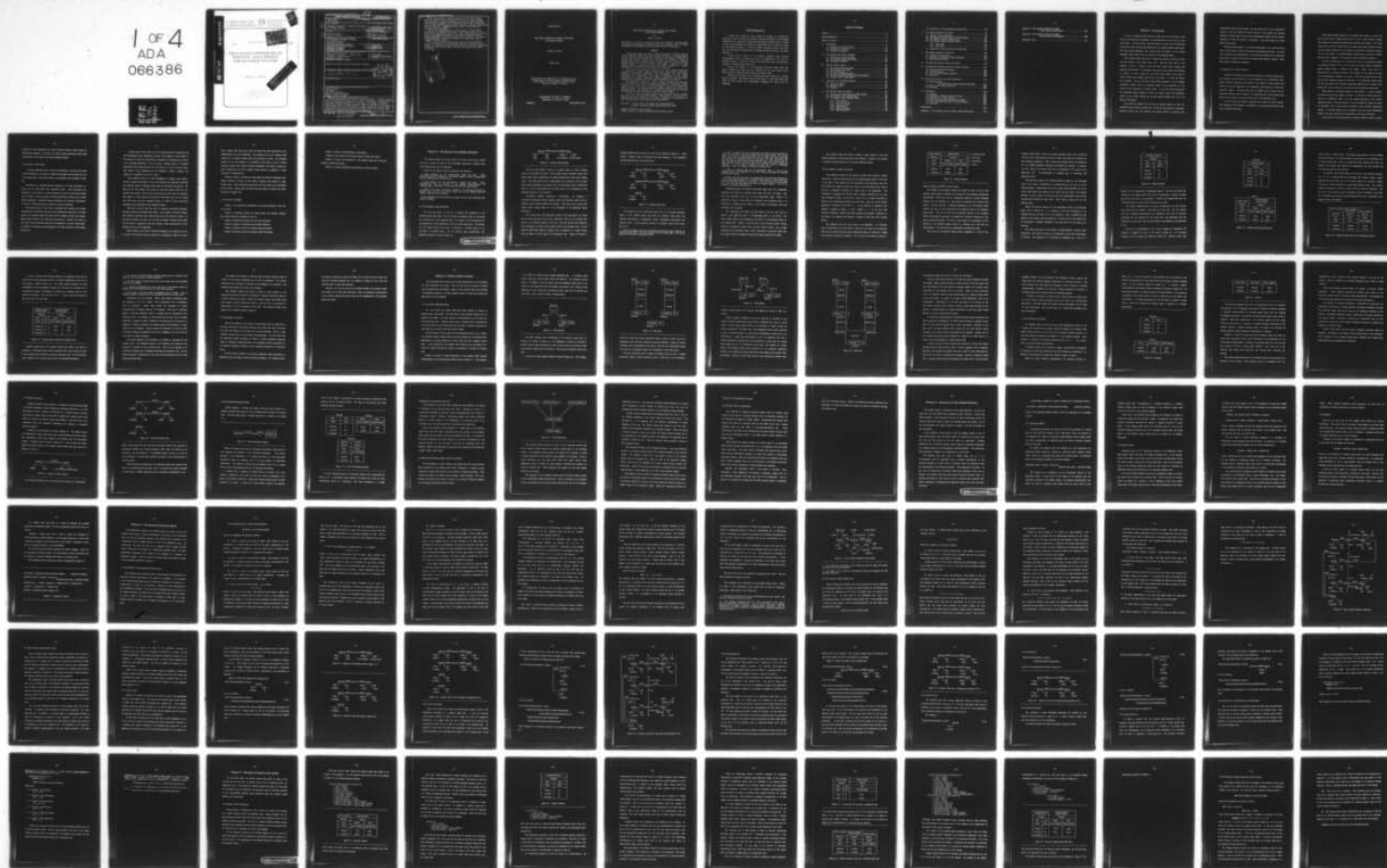
MIT/LCS/TR-206

N00014-75-C-0661

NL

1 OF 4
ADA
066386

4-11-79



AD A066386

DDC FILE COPY

LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

12
B.S.

MIT/LCS/TR-206



EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS FOR DATABASE SYSTEMS

Sheldon A. Borkin



This research was supported by the
Advanced Research Projects Agency
of the Department of Defense
and was monitored by the Office of Naval Research
under Contract No. N00014-75-C-0661

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-206	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Equivalence Properties of Semantic Data Models for Database Systems,	5. TYPE OF REPORT & PERIOD COVERED Ph.D. Thesis - Jan. 1979	6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-206
7. AUTHOR(s) Sheldon A. Borkin	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0661	9. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Department of the Navy Information Systems Program Arlington, VA 22217
10. PERFORMING ORGANIZATION NAME AND ADDRESS MIT/Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139	11. REPORT DATE January 1979	12. SECURITY CLASS. (of this report) Unclassified
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency Department of Defense 1400 Wilson Boulevard Arlington, VA 22209	13. NUMBER OF PAGES 315	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Doctoral thesis,		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) database systems data models data model equivalence semantic relation data model semantic graph data model		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A data model defines the types of structures present in a database and the types of operations which may be used to alter the database. An understanding of data model equivalence properties is necessary if one wishes to implement a system which presents different users with views of a database in terms of differing data models or which provides a common interface to several database systems defined in terms of different data models. Requisites for this understanding are formal definitions of the involved concepts and a formal framework in which different data models can be compared.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DDC
RECEIVED
MAR 26 1979
C

409 648

20.

This thesis presents formal definitions of the terms database, operation, operation type, application model and data model. Database state equivalence, operation equivalence, application model equivalence and data model equivalence are distinguished. Three types of application and data model equivalence are defined - isomorphic, composed operation and and state dependent. Implementation implications of these different equivalences are discussed.

This definitional framework is used to explore the equivalence properties of two semantic data models. Semantic data models are data models whose structures are meant to have clear interpretations in terms of the the applications being modelled. The semantic graph and semantic relation data models are formally defined. The definition of the semantic relation data model includes the definitions of constraints and the semantic relational algebra.

It is proved that the semantic graph and the restricted semantic relation data models are state dependent equivalent. Observations on the network vs. relational data model "controversy" are presented. Suggestions for applications of this research include a dual semantic data model interface.

Form with handwritten 'A' and checkboxes:

Write Section	<input type="checkbox"/>
Diff Section	<input type="checkbox"/>
UNCLASSIFIED	
BY DISTRIBUTION/AVAILABILITY CODES	
Dist	or SPECIAL

Handwritten 'A' in the bottom left box.

MIT/LCS/TR-206

**EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS
FOR DATABASE SYSTEMS**

Sheldon A. Borkin

January 1979

**This research was supported by the Advanced Research
Projects Agency of the Department of Defense and was
monitored by the Office of Naval Research under
Contract No. N00014-75-C-0661**

**MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE**

CAMBRIDGE

MASSACHUSETTS 02139

EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS
FOR DATABASE SYSTEMS

by

Sheldon A. Borkin

This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on 2 January 1979 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

A *data model* defines the types of structures present in a database and the types of operations which may be used to alter the database. An understanding of *data model equivalence properties* is necessary if one wishes to implement a system which presents different users with views of a database in terms of differing data models or which provides a common interface to several database systems defined in terms of different data models. Requisites for this understanding are formal definitions of the involved concepts and a formal framework in which different data models can be compared.

This thesis presents formal definitions of the terms *database*, *operation*, *operation type*, *application model* and *data model*. *Database state equivalence*, *operation equivalence*, *application model equivalence* and *data model equivalence* are distinguished. Three types of application and data model equivalence are defined - *isomorphic*, *composed operation* and *state dependent*. Implementation implications of these different equivalences are discussed.

This definitional framework is used to explore the equivalence properties of two *semantic data models*. Semantic data models are data models whose structures are meant to have clear interpretations in terms of the applications being modelled. The *semantic graph* and *semantic relation data models* are formally defined. The definition of the semantic relation data model includes the definitions of *constraints* and the *semantic relational algebra*.

It is proved that the semantic graph and the *restricted semantic relation data models* are state dependent equivalent. Observations on the network vs. relational data model "controversy" are presented. Suggestions for applications of this research include a *dual semantic data model interface*.

Keywords: database systems, data models, data model equivalence
semantic relation data model, semantic graph data model

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Professor of Electrical Engineering and Computer Science

Acknowledgements

I thank Jack Dennis for many things: for making the Computation Structures Group a friendly and hospitable research environment, for providing feedback on my research, for making financial support available when I needed it, and most importantly, for having the implicit trust that I would not abuse the privileges he granted. I thank my readers, Professors Szolovits and Elias, for their constructive comments and for helping me make this thesis a readable document.

I thank Herbert Weber for his constant, sometimes long distance, encouragement and for providing me with an introduction to the area of database systems. I thank Ken Weng for his many hours spent in discussions acting as a sounding board for my ideas.

I thank the members of the Computation Structures Group, both the present members and those who have graduated over the past several years. They have been both professional colleagues and close friends.

I thank my wife, Sandy, for her patience and support which were always taken for granted. She makes every day worth living. I thank my parents and sister for their support and encouragement throughout the many years of schooling.

Table of Contents

Abstract.....	2
Acknowledgements	3
Table of Contents	4
1. Introduction.....	7
1.1 Motivation for Thesis Research	8
1.2 Concerns of this Thesis	10
1.3 Overview of Chapters.....	12
2. The Semantic Data Models: Examples	15
2.1 The Semantic Graph Data Model	15
2.2 The Semantic Relation Data Model	19
2.3 Equivalence Properties	27
3. Relevant Work of Others	29
3.1 The Codasyl DBTG Data Model	29
3.2 The Relational Data Model.....	35
3.3 Semantic Networks	39
3.4 The Entity-Relationship Model.....	41
3.5 Multiple Data Model Database System Architecture.....	43
3.6 Previous Work on Equivalence	46
4. Framework for Data Model Definition	49
4.1 Application Models.....	50
4.2 Operation Types.....	51
4.3 Databases	53
5. The Semantic Graph Data Model.....	55
5.1 The Semantic Graph Application Model Schema	55
5.2 The Semantic Graph Database State.....	62
5.3 The Semantic Graph Operation Types.....	68
5.3.1 Insert-ind-entity	69
5.3.2 Insert-ind-association	72
5.3.3 Insert-semantic-unit	75
5.3.4 Delete-ind-entity	77
5.3.5 Delete-ind-association.....	78
5.3.6 Delete-semantic-unit	79

5.4	Definitions of Valid Operations and Valid States	80
6.	The Semantic Relation Data Model	85
6.1	Semantic Relation Statements	85
6.2	The Semantic Relation Application Model Schema	95
6.3	The Semantic Relation Database State	97
6.4	The Semantic Relation Operation Types	100
6.4.1	Insert-tuples	100
6.4.2	Delete-tuples	114
6.5	Definitions of Valid Operations and Valid States	122
7.	Data Model Equivalence: Definitions	125
7.1	Database State Equivalence	125
7.2	Operation and Application Model Equivalence	130
7.3	Data Model Equivalence	137
8.	The Restricted Semantic Relation Data Model	139
9.	The Restricted Semantic Relation Data Model: Constraints	161
9.1	The Semantic Relational Algebra	162
9.2	Constraint Types	163
9.3	The Constraint Definition Algorithm	166
9.4	Discussion	195
10.	Constructive Proof of Data Model Equivalence	213
10.1	State Equivalence for the Restricted Semantic Relation Data Model	214
10.2	Proof of Data Model Equivalence	222
10.3	Discussion	245
11.	Conclusions	249
11.1	Summary	249
11.2	Relational vs. Network: Some Observations	252
11.3	Foundation for a User Interface	254
11.4	Inferential Knowledge Representation Systems	260
11.5	Other Applications and Areas for Future Research	261
11.6	Conclusion	266
	Bibliography	269
	Appendix 1. The Semantic Graph Data Model: Formal Specifications	275

Appendix 2. The Semantic Relation Data Model:	
Formal Operation Specifications.....	291
Appendix 3. The Semantic Relation Data Model:	
Formal Constraint Specifications.....	299
Biographic Note	311

Chapter 1 - Introduction

A user of a database system always has some view of the structure of the data in the database and knowledge of the operations which can be used to alter the database. The types of data structures visible to the user and the operations allowed on these structures are determined by the database system's *data model*. The data model determines what features must be present in the various languages which make up the user interface to the database system.

The database system may just be a simple file system in which the user interacts directly at the storage device level. Then the user's view of the database is that of a group of files each with certain record formats and indexing characteristics. The applicable operations are the insertion and deletion of records. In such a system, the data model would specify what types of record formats and indexing characteristics are allowed. It would also define the operations. We will say that a data model determines a set of possible *application models* where an application model is the description of the structures and operations in a specific system. In the case of the file system, the application model describes a specific file system where this application model is one of those allowed by the data model describing the set of all possible file systems.

In an effort to simplify the user's view of database systems, to allow for portability across different machines and to allow for easier growth of databases, database systems have been designed with greater degrees of *physical data*

independence [Date and Hopewell]. The data models used in data independent systems let the user define the logical structure of the database and perform operations without regard to lower level implementation details and the specific machine on which the system is implemented. Such data models are called *logical data models*.

Existing implementations of such data models present to the user structures such as trees, tables, or linked lists along with appropriate operations. The most often compared data models ([Nijssen 74] and [Date and Codd]) are the *relational data model* of [Codd 70,72,74] and the *network data model* of [DBTG]. These data models are described in Chapter 3.

1.1 Motivation for Thesis Research

Different data models may seem most appropriate for different applications. For example, one data model may be most appropriate for an application which makes numerous queries but few updates to the database, while another data model may be more appropriate for an application which makes few queries but numerous updates. One data model may be suitable for the person who is intimately familiar with the details of the application area, while another data model may be easier to use for the person naive to the application area.

In recent years, the number of proposed data models has grown rapidly. The samplings of data models in [Kerschberg et. al.] and [Senko] mention over thirty different data models.

Given these possible preferences for different data models, we must ask what can be done if different users want to see the "same" database through the perspective of application models based on different data models? We would want users to be able to make queries or updates using either application model with an update performed via one model to be reflected in subsequent queries via the other model. The ANSI/X3/SPARC database system architecture [ANSI] proposes such a capability. This proposal is further discussed in Chapter 3.

In such an architecture, it is desired that a user of one application model is completely unaware of the fact that others are using different application models possibly based on different data models. Clearly, there must be constraints on the application models so that updates via one application model can be represented as updates in the others. If this can be done, then we will say that the application models are *equivalent*. If all application models based on one data model have equivalent application models based on another data model, and *vice versa*, then we will say that the data models are equivalent.

These notions of equivalence appear in other contexts. It may be desired to transfer data from one system to another. In this case we must find out if there is an application model in the new system equivalent to the current application model. If the data models on which the application models are based are equivalent, then it will always be possible to find equivalent application models. If the data models are not equivalent, then only certain application models in one data model will have equivalents in the other data model.

In the context of distributed systems, it might be desired to have a central

interface to data distributed over several different database systems based on different data models. In this case, the central interface application model must be equivalent to the "sum" of the other application models.

1.2 Concerns of this Thesis

To clearly understand what is meant by "equivalence", one must have clear formal definitions of the relevant concepts and examples demonstrating the use of the definitions. This thesis presents such definitions and examples of their use.

Prerequisites to formally defining equivalence are formal definitions of terms such as "data model" and "application model". Such definitions are presented in this thesis as a formal framework in which various data models can be defined. Given this ability to define data models, concepts such as "data model equivalence", "application model equivalence" and "operation equivalence" are defined. These definitions are based on the work in [Borkin].

The major portion of this thesis is concerned with applying this formal framework to the definition and study of the equivalence properties of two data models, the *semantic graph data model* and the *semantic relation data model*. Being loosely related to the relational and network data models, the study in this thesis should shed some light onto the relational vs. network "controversy" in addition to providing an understanding of the issues involved in data model equivalence.

A major claim of this thesis is that the various notions of equivalence are most meaningful when consideration is given in the design of a data model to the manner in which the data model can represent the information of concern to the intended application. If we are using a database system to represent information about some portion of the real world, it is not at all clear which data model is most appropriate for the application. Should a person, for example, be represented as a tree or a table?

Such problems have led to the investigation of *semantic data models*. These are logical data models in which the structures and operations permitted are explicitly meant to represent certain types of real-world information. We will use the term *syntactic data models* for those data models which are not semantic data models. The two data models defined in this thesis are semantic data models. If two semantic data models are equivalent, then we are saying that they have the same "expressive power" in terms of what real-world situations can be described by their application models.

Note that we do not claim that there is a perfectly well-defined boundary between syntactic and semantic data models. In one sense, all formally defined data models are syntactic since formal systems are in the end pure syntax. In the sense in which we wish to use the term, we regard a semantic data model as one which is conducive to the user having a direct interpretation of the database in terms of the application.

Many workers in the field of artificial intelligence have explored the use of semantic data models which are appropriate for applications which are much

more complex than those with which this thesis will deal ([Szolovits et. al.], [Mylopoulos et. al.] and [Simmons]). The intended use of such complex data models are in computer systems which are inferential in nature. The intended purpose of the data models to be presented in this thesis is not to create computer systems with "greater inferential capabilities", but rather to allow the easier organization and use of database systems similar in capability to those currently in widespread use.

However, many of the features which make the artificial intelligence data models semantic in nature are also useful in semantic data models of more limited scope. These features have been used in the data models to be presented in this thesis. Similar ideas are behind the data models of [Chen] and those described in [Nijssen 76].

1.3 Overview of Chapters

Chapter 1 has presented an introduction to the topics discussed in the rest of the thesis.

Chapter 2 informally presents the semantic graph and semantic relation data models along with examples of their use.

Chapter 3 discusses the relevant work of other researchers.

Chapter 4 presents a framework for data model definition.

Chapter 5 describes in detail the semantic graph data model.

Chapter 6 describes in detail the semantic relation data model.

Chapter 7 presents formal definitions of equivalence.

Chapters 8 and 9 define the *restricted semantic relation data model*.

Chapter 10 proves the equivalence of the semantic graph and restricted semantic relation data models.

Chapter 11 contains conclusions and directions for future research.

Chapter 2 - The Semantic Data Models: Examples

For simple examples of the data models to be used in this thesis, consider the case in which we want to store information concerning a machine shop.

The following describes this application:

1. There are two types of *entities*: "employees" and "machines".
2. Each employee has two *characteristics*: "name" and "age". These characteristics must be elements of the domains "names" or "years", respectively. No two employees have the same name.
3. Each machine has two characteristics: "number" and "type". These characteristics must be elements of the domains "serial-numbers" or "machine-types", respectively. No two machines have the same number.
4. There is one type of *association*: "operation". The operation association is defined with two roles: the "agent" which must be an employee and the "object" which must be a machine.
5. Each machine must have exactly one operator, but not every employee need operate a machine.

2.1 The Semantic Graph Data Model

The first data model we will use to represent this application is the *semantic graph data model*. We will define an application model as consisting of a *schema* and a set of operation types. The schema is the representation of the application in terms appropriate to the data model being used. The schema for the semantic graph data model is expressed as a labeled graph or its equivalent set notation form. For the situation under consideration, the appropriate schema is shown in Figure 2.1.

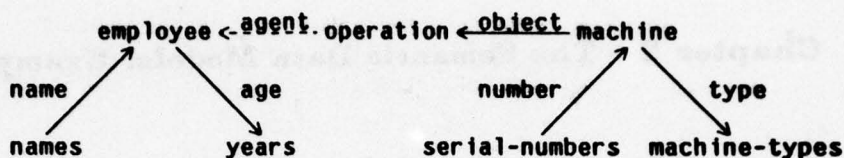


Figure 2.1 - Semantic Graph Schema

There are three kinds of nodes in the graph: *entity set names*, *domain names* and *association set names*. *Role set edges* connect association nodes with entity nodes while *characteristic set edges* connect entity nodes with domain nodes. Solid lines used as role set edges signify that a particular type of entity must always participate in a particular role in an association while a dotted line used as a role set edge means that the entity may be in an association of the specified type but it is not required.

Arrowheads represent uniqueness or functionality. For example, a functional relationship between employee names and employees means that an employee name uniquely identifies an employee. With these facts in mind, the reader should compare Figure 2.1 to the description of the application stated previously.

The current state of the application consists of the specification of: What are the names and ages of the current employees? What are the numbers and types of the machines currently in the shop? And who operates each machine? This information is stored in the database system as the *database state*. In the semantic graph data model the database state is represented as a graph which has one object for each object in the application state. Figure 2.2 shows a

possible database state as seen by a user via the schema in Figure 2.1. This shows a situation with two machines and three employees. Two employees operate machines while the third does not.

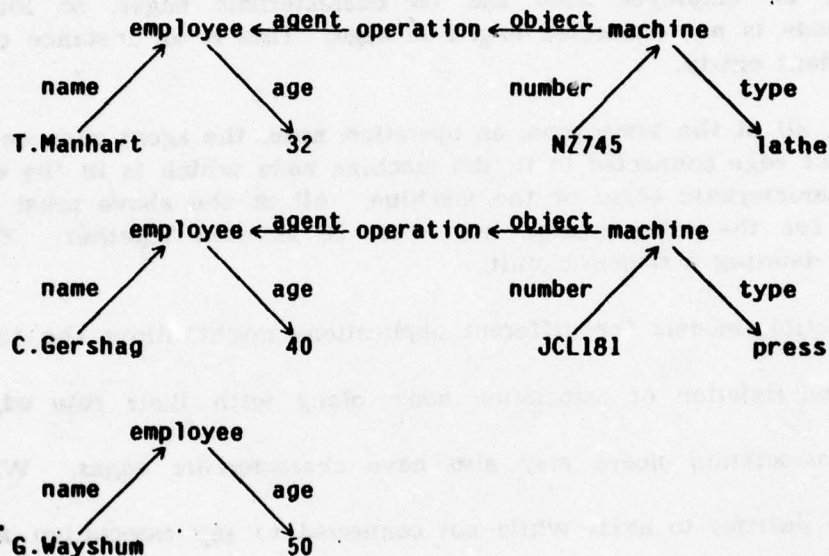


Figure 2.2 - Semantic Graph State

The second part of the application model is the set of allowed operation types. In the semantic graph data model, the operation types allow the insertion and deletion of *independent entities*, *independent associations* and *semantic units*. In this particular situation, the operation types would allow the user to:

1. Insert an employee node and corresponding characteristic edges checking to make sure that there is no other employee with the same name. This is an instance of inserting an independent entity.

2. Insert a machine node and corresponding characteristic edges checking to make sure that there is no other machine with the same number. And, *at the same time*, insert an operation association node, an object role edge connecting the machine and operation nodes, and an agent role edge connecting the operation node to some already existing employee node. These actions must be done together because the schema states that no machine can exist without an operator. So if we insert a machine, we must also insert an operation association. This is an instance of inserting a semantic unit.

3. Delete an employee node and its characteristic edges, so long as the employee node is not connected to a role edge. This is an instance of deleting an independent entity.

4. Delete, all at the same time, an operation node, the agent edge connected to it, the object edge connected to it, the machine node which is in the object role and the characteristic edges of the machine. All of the above must be deleted as a unit for the same reason they must be inserted together. This is an instance of deleting a semantic unit.

Application models for different applications might allow the independent insertion and deletion of association nodes along with their role edges. Also note that association nodes may also have characteristic edges. While it is possible for entities to exist while not connected to any association node, it is necessary for appropriate role entities to exist before an association node and role edges can be inserted.

The fundamental idea behind this data model is that for every type of object in the application there is a corresponding object in the schema. For every unique object in the application state there is a corresponding object in the database state. "Object" is used not just to mean "physical objects" which would be represented as entity nodes, but also "abstract objects" such as ages represented as characteristic nodes, "events" represented as association nodes, and the various types of associations between these objects represented as edges.

The semantic graph data model is similar in some respects to the data models presented by [Chen] and [Schmid and Swenson]. Chapter 5 will present a complete formal description of the semantic graph data model.

2.2 The Semantic Relation Data Model

The second data model we will consider in detail is the *semantic relation data model*. While the basic idea behind the semantic graph model is that there is one object in the database state for each object in the application state, the idea behind the semantic relation data model is that the database state consists of sets of *statements* describing the current state of the application. The statements are organized into *relations* where a relation represents the set of all true statements fitting a certain form. Figure 2.3 is a relation describing a portion of the hypothetical machine shop world state. Each row in the relation represents a statement of the form: "____ is the name of an employee who operates a machine of type ____ with number ____." So the row with elements "T.Manhart", "NZ745" and "lathe" represents the sentence: "T.Manhart is the name of an employee who operates a machine of type lathe with number NZ745."

The ordering of the rows and columns is not significant. No statement may be duplicated in the same relation. That is, no two rows may be identical. Note that this differs from the usual mathematical notion of relations in which the ordering of columns is significant. We can allow this departure because a

operate:agent	operate:object		← predicate:case pair
employee	machine		← entity type
name	number	type	← characteristic
names	serial-numbers	machine-types	← domain
T.Manhart	NZ745	lathe	
C.Gershag	JCL181	press	

Figure 2.3 - Semantic Relation

unique heading is specified for each column.

The design of the semantic relation data model (as well as that of the semantic graph model) is influenced by the notion of *case grammars* ([Fillmore] and [Bruce]). This states that the "underlying" meaning of natural language sentences can be expressed as a verb phrase (a *predicate*) plus several noun phrases - one for each case. A predicate corresponds to the verb phrase which describes the type of event represented by an association name in the semantic graph model. That is, we will use the predicate "operate" in the semantic relation data model to describe in a statement the "operation" association in the semantic graph data model. The role an entity plays in an association is thus specified in the semantic relation model by a predicate and a case such as "operate:agent". We will call such a specification a *predicate:case pair*.

Each case in the semantic relation model corresponds to a role in the

semantic graph model. There is no general agreement about what constitutes the "best" set of cases (and hence roles) or which cases (roles) are necessary for each predicate (association). Many authors have proposed their own guidelines for this problem ([Fillmore], [Martin] and [Schank] for example). In general, the most useful set of cases (roles) must be determined by the needs of the application area. Its determination is considered part of specifying the application model.

Note that each column in the example relation of Figure 2.3 has associated with it four items: a specification of a predicate:case pair, the type of entity being described, a characteristic name, and a domain name specifying the set of values from which each element in the column must be drawn. For display purposes we may "factor" the role and entity type specification when several characteristics describe the same entity. This is done in Figure 2.3 for the machine entity.

Figure 2.3 does not contain all of the information shown in the semantic graph state of Figure 2.2. It is missing the fact that G.Wayshum is an employee and also the ages of all of the employees. This is remedied by the addition of the relation shown in Figure 2.4. The first row in this relation can be interpreted as meaning: "T.Manhart is the name and 32 is the age of an employee."

The predicate:case pair in this relation, "be employee:object", deserves some explanation. The objects of interest in the application can be either associations or entities. The existence of an association is represented by a row in a

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Figure 2.4 - Semantic Relation

relation with the association's corresponding predicate. This will also assert the existence of the entities which have roles in the association. However, entities may exist which are in no associations. Therefore the predicate:case pair "be entity-type:object" is used to state the existence of entities.

The example relations given so far give the impression that each relation deals with only one predicate. This need not be the case. We will require that for a complete representation of an application, there must be relations containing the "be" predicate for each entity type. This guarantees that the existence of every entity will be stated whether it participates in an association or not.

This can be accomplished in the current example by modifying the relations of Figures 2.3 and 2.4 into those of Figure 2.5. For reference purposes, we have named the relations in Figure 2.5. (Relation names will

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 2.5 - Complete Semantic Relation State

always begin in capital letters.) The important new feature is that the second and third columns of the Operate relation are now shown to be describing a set of predicate:case pairs. Since it is stated that every machine must have an operator, the columns that describe the "operate:object" pair also describe the "be machine:object" pair for machines.

It is possible, and would certainly be the case in a more complex situation, that the set of predicate:case pairs describing some column contains several predicate:case pairs where none of these pairs concern "be" predicates. Figure 2.6 shows a possible relation for a more complex machine shop. This relation contains information on both the "operate" and the "supervise" predicates. The first row in this relation would be interpreted to mean: "T.Manhart is the name of an employee who operates machine number NZ745 and is supervised by an employee named G.Wayshum."

supervise:agent	supervise:object operate:agent	operate:object
employee	employee	machine
name	name	number
names	names	serial-numbers
G.Wayshum	T.Manhart	NZ745
G.Wayshum	C.Gershag	JCL181

Figure 2.6 - Semantic Relation With Set of Predicate:Case Pairs

In such a relation with multiple predicates, the application state may be such that not every predicate:case pair in a relation is applicable to each row in the relation. Consider Figure 2.7. This single relation contains the same information as the two relations in Figure 2.5. The last row in Figure 2.7 is interpreted as saying: "G.Wayshum is the name of an employee, age 50, who operates no machine." We will refer to the "---" used to show the absence of any other value as a *null* value.

be employee:object operate:agent		be machine:object operate:object	
employee		machine	
name	age	number	type
names	years	serial-numbers	machine-types
T.Manhart	32	NZ745	lathe
C.Gershag	40	JCL181	press
G.Wayshum	50	---	---

Figure 2.7 - Single Semantic Relation For Complete State

Another essential part of the semantic relation data model is the idea of constraints. Constraints specify certain conditions which must always be true of the *relational state* describing a particular application state. For the relations used in Figure 2.5, we would require at least the following constraints:

1. The names in the first column of relation *Operate* must be a subset of the names in the first column of relation *Employees*.

2. The first column of *Operate* may have no null values since every machine must have an operator.

3. A specific serial-number may occur only once in the second column of *Operate* since each machine may have no more than one operator.

4. The names in the first column of *Employee* must be unique. This is required because we stated that no two employees may have the same name.

Constraints serve two purposes. Firstly, they provide information about the meaning of a row in a relation. This is particularly true of constraints, such as constraint 4 above, which specify the uniqueness of certain characteristics in relations with the "be" predicate. This type of constraint enables a relational statement to refer to a specific entity or association in the application state. For example, we stated earlier that the first row of relation *Employees* in Figure 2.5 means: "T.Manhart is the name and 32 is the age of an employee." However, nothing in this sentence rules out the possibility of there being two such employees. With the addition of constraint 4, we can say that the correct statement represented by the row is: "T.Manhart is the name and 32 is the age of a *unique* employee."

The second purpose of the constraints is in relation to operations on the database state. The operations allowed in the relational data model are the insertion or deletion of sets of relation rows. This is to be thought of as inserting or deleting sets of statements concerning the application state. But an update operation is allowed only if all of the stated constraints are true of the resulting database state.

We consider the schema of a particular semantic relation database model to consist of the relation definitions and the set of constraints. The relation definitions can be thought of informally as the headings of the relations. The database state consists of the rows in the relations.

The semantic relation data model is similar in some respects to the relational data model of [Codd 70,71,72a,72b,74]. However, Codd's data model is a simple syntactic data model. Others have proposed similar data models with some semantic features ([Chen] and [Hall et. al.]), but not in the depth or completeness of the one presented in this thesis. The semantic relation data model will be formally defined in Chapter 6.

2.3 Equivalence Properties

While the examples in this chapter are quite simple, they are sufficient to introduce the kinds of equivalence properties with which this thesis will deal. We claim that Figure 2.5 and Figure 2.2 are *state equivalent*. That is, they represent identical states of the machine shop application. We also claim that the application model represented by Figure 2.1 (with appropriate operation types) is *application model equivalent* to the application model represented by the relational headings of Figure 2.5 (with appropriate constraints and operation types).

As will be seen in Chapter 7, the claim of application model equivalence is dependent upon the ability to derive *equivalent operations*. For example, given

an operation inserting an entity into Figure 2.2, we must be able to derive an operation inserting statements into the relations of Figure 2.5 such that the resulting states are also state equivalent.

However, we will also see that the semantic relation and semantic graph data models are not *data model equivalent*. There are applications representable in the semantic relation data model which are not representable in the semantic graph data model.

Chapter 3 - Relevant Work of Others

We will present the relevant work of other researchers by first discussing the most influential data models. Then we will see how semantic concepts were introduced into data models. Lastly we will consider other work relating to equivalence properties. For more complete surveys of various data models see [Kerschberg et. al.] or [Senko].

3.1 The Codasyl DBTG Data Model

We will discuss the Codasyl DBTG data model [DBTG] as typical of syntactic graph data models. This data model is more generally referred to as a *network* data model. The basic methods of structuring data in this model are by *records* and "*sets*". Different record types are declared with a specification of the *data items* in each record type and what kind of values are allowed for each item (e.g. a Cobol or PL/1-like "picture" clause).

The set concept is not the same as the mathematical notion of set. Rather a set specifies "links" between different record occurrences in the database state. Specifically, a set type consists of an owner record type and a member record type. A set occurrence in the database state consists of one occurrence of a record of owner type and any number of occurrences of records of member type.

Figure 3.1 shows a partial specification of one possible DBTG schema corresponding to the semantic graph schema shown in Figure 2.1. This diagram

is a variant of a *data structure diagram* [Bachman 69]. It specifies three record types and two set types, Emp-op and Mach-op. The Operation record type is a member of both set types while the Employee record type is the owner of the Emp-op set and the Machine record type is the owner of the Mach-op set type. Note that the Operation record type does not include any data items. Such a record is called a "linking record".

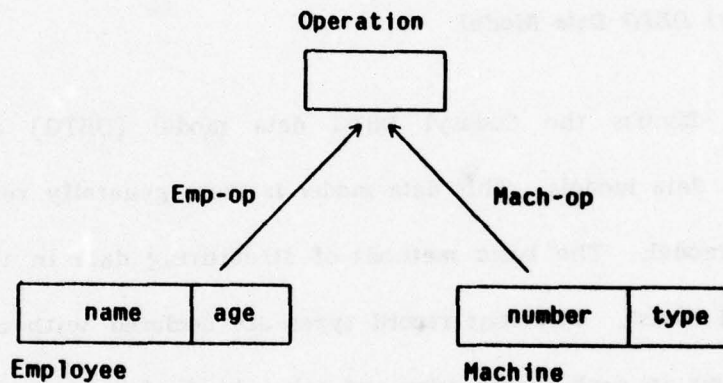


Figure 3.1 - DBTG Schema

The DBTG database state corresponding to the semantic graph state in Figure 2.2 is shown in Figure 3.2. Set membership is shown by drawing a directed cycle of links connecting the owner record of each set occurrence with each member record of the set occurrence. The set type is specified within the cycle.

A portion of a more complex schema is shown in Figure 3.3. This schema

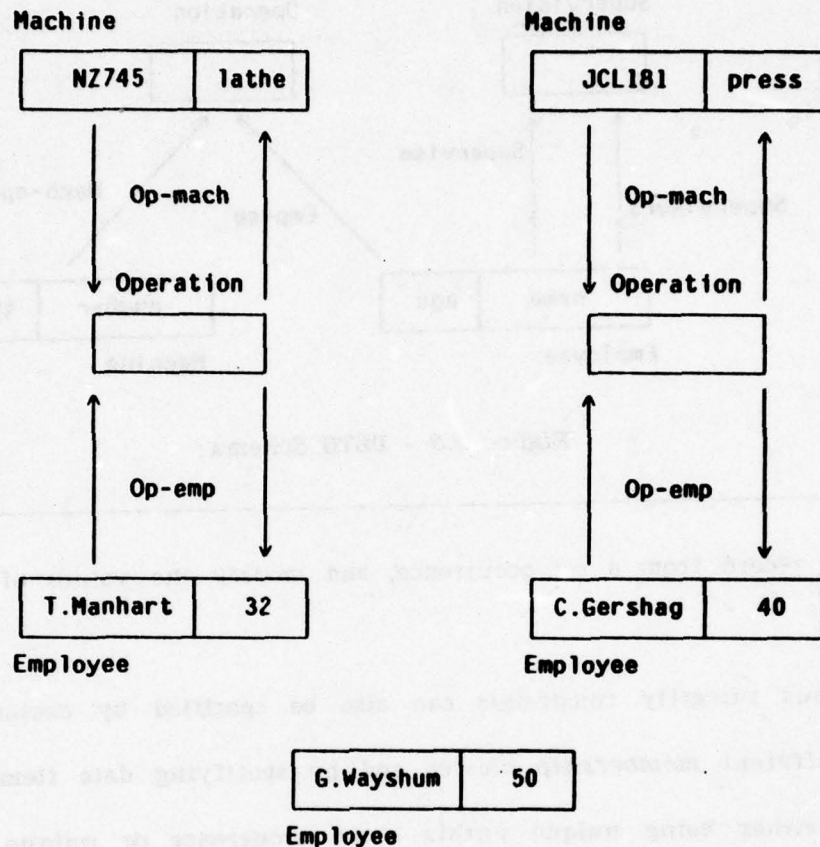


Figure 3.2 - DBTG State

includes a record type schema Supervision which is meant to show, via its set membership, the supervision relationships between employees. Figure 3.4 shows a database state corresponding to this schema. Note that two supervision records are owned by the Employee record with "G.Wayshum" as the name value.

The basic operations used to change the database state are: *store* a record occurrence, *delete* a record occurrence, *insert* a record into a set occurrence,

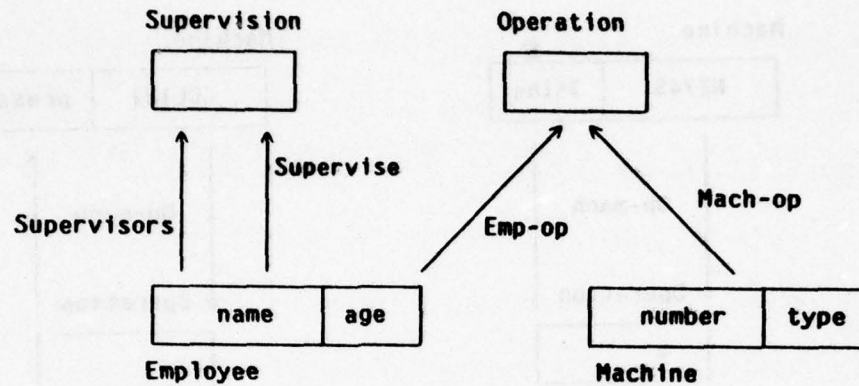


Figure 3.3 - DBTG Schema

remove a record from a set occurrence, and *modify* the values of items in a record.

Various integrity constraints can also be specified by declaring in the schema different *membership classes* and by specifying data items in record types as either being unique within a set occurrence or unique within all record occurrences of the record type. The membership classes specify whether or not a record occurrence of a given type is *automatically* inserted into a set occurrence of a given type and whether or not it is *mandatory* that once a record occurrence is inserted into a set occurrence of a given type, the record must always be in a set occurrence of that type. For example, we would declare the Machine record type in Figure 3.1 as an automatic mandatory member of the Mach-op set type stating that every machine must always have an operator. Note that if the owner record of a set occurrence is deleted, then

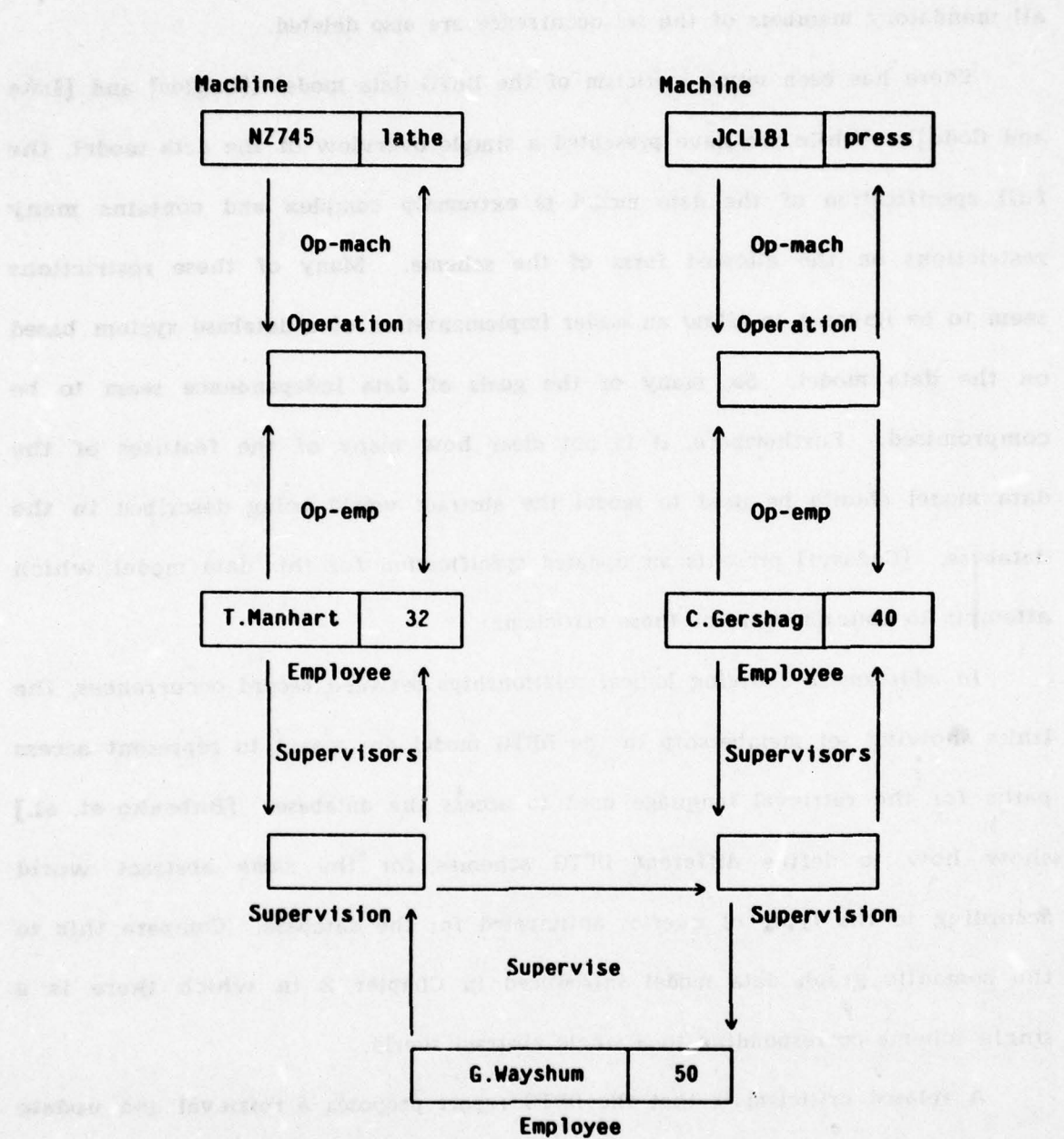


Figure 3.4 - DBTG State

all mandatory members of the set occurrence are also deleted.

There has been much criticism of the DBTG data model ([Engles] and [Date and Codd]). While we have presented a simple overview of the data model, the full specification of the data model is extremely complex and contains many restrictions on the allowed form of the schema. Many of these restrictions seem to be imposed to allow an easier implementation of a database system based on the data model. So, many of the goals of data independence seem to be compromised. Furthermore, it is not clear how many of the features of the data model should be used to model the abstract world being described in the database. [Codasyl] presents an updated specification for this data model which attempts to counter some of these criticisms.

In addition to showing logical relationships between record occurrences, the links showing set membership in the DBTG model are meant to represent access paths for the retrieval language used to access the database. [Bubenko et. al.] show how to define different DBTG schemas for the same abstract world according to the types of queries anticipated for the database. Compare this to the semantic graph data model introduced in Chapter 2 in which there is a single schema corresponding to a single abstract world.

A related criticism is that the DBTG report proposes a retrieval and update language based on "one-at-a-time" access. In such a language a user "navigates" [Bachman 73] through the database state from record to record following the links with the help of a procedural host language. However it should be noted that a network structure need not necessarily be accessed with a "one-at-a-time"

language. [McGee 75] and [Deheneffe and Hennebert] present proposals for higher-level set oriented languages appropriate for a network structure. The basic problem with these higher-level languages is that of finding an efficient implementation. [Bachman and Daya] have recently proposed an extension of the DBTG model which contains notions of modelling semantics.

If links in the schema are restricted so as to require that the database state is a set of trees of record occurrences, then this is said to be a *hierarchical data model*. The most widely used database system, IMS [McGee 77], is based on such a data model. Both "one-at-a-time" and higher-level languages exist for IMS [Fehder].

3.2 The Relational Data Model

To eliminate from the user's view the access dependencies present in the network and hierarchical data models, [Codd 70] proposed the relational data model (often referred to in this thesis as the syntactic relational model). In this data model, the database state consists of a set of named *relations*. Each relation is a subset of the Cartesian product space $(D_1 \times D_2 \times \dots \times D_n)$ where the D_i are sets of values called *domains*.

A relation is then a set of n-tuples, or *tuples*. The relations are displayed in tabular form where the orderings of rows and columns are insignificant. A column is identified by the domain from which its values are drawn.

Figure 3.5 shows relations corresponding to the semantic relations of

Figure 2.5. It can be seen that in this particular case, the removal of the semantic information in the top three rows of the semantic relations' headings results in the syntactic relation of Figure 3.5. A syntactic relation corresponding to the semantic relation of Figure 2.6 is shown in Figure 3.6. To distinguish between two columns defined on the same domain, a *role qualifier* is appended to the domain name as shown in Figure 3.6 by "oper.names" and "super.names". A further generalization of the data model is defined in [Codd 72a] in which all columns have *attribute* names in addition to a domain specification.

Employees

names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Operate

names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 3.5 - Relations

super.names	oper.names	serial-numbers
G.Wayshum	T.Manhart	NZ745
G.Wayshum	C.Gershag	JCL181

Figure 3.6 - Relation

The retrieval languages used with the relational data model are generally set oriented. We can distinguish three types of relational set-oriented languages - relational calculus [Codd 71], relational algebra [Codd 72b] and mapping [Chamberlin and Boyce]. The relational calculus retrieves tuples which satisfy a predicate calculus-like qualification. The relational algebra retrieves tuples which are the result of a sequence of relational operations performed on the database relations. Mapping languages specify the tuples to be retrieved by taking subsets of chosen relations and compositions of relations.

However, it should be noted that several relational systems are based on lower-level one-tuple-at-a-time retrieval ([Astrahan et. al.], [Stonebraker et. al.] and [Schmid and Bernstein]). [Astrahan et. al.] actually allow the user to access the relations via such a tuple-at-a-time interface. As in the case of the network data model, both "high-level" and "one-at-a-time" interfaces are possible.

The update operations allowed in the relational model are the insertion and deletion of sets of tuples. These operations must be in accordance with the

maintenance of some subset of each relation's columns as the *key* of the relation. That is, a tuple can be uniquely identified by the values in these columns.

[Codd 72a] proposes "normal forms" for relations to prevent "update anomalies". These anomalies correspond to database states which do not represent possible states of the abstract world being modelled. The definition of the normal forms depends on the analysis of *functional dependencies*.

The concept of functional dependencies is understood by noting that the definition of the key of a relation specifies a functional dependency between the attributes which make up the key and the remaining attributes in the relation. Functional dependencies specify certain types of semantic features of the abstract world. However, as pointed out in [Bernstein], they are not sufficient to capture all of the semantic information necessary to maintain the database in a consistent state.

[Codd 70] also noted that other forms of integrity constraints might be desirable. An update to the database would be allowed only if all constraints were "satisfied" by the resulting database state. [Hammer and McLeod] and [Stonebraker] have investigated such constraints.

3.3 Semantic Networks

Semantic networks were introduced as a method of representing knowledge in artificial intelligence research ([Quillian], [Simmons], [Szolovits et. al.] and [Mylopoulos et. al.]). [Schmid and Swenson] use a simple semantic network data model to explain what the definition of normal form relations [Codd 72a] means in terms of the abstract world being modelled. They also show how constraints other than functional dependencies are necessary to maintain semantic integrity.

A schema in this data model is shown in Figure 3.7. This figure defines the characteristic object types names, years, serial-numbers and machine-types, the independent object types employee and machine, and the association operate. A database state is shown in Figure 3.8. It can be seen that these figures are quite similar to the semantic graph schema and state shown in Figures 2.1 and 2.2.

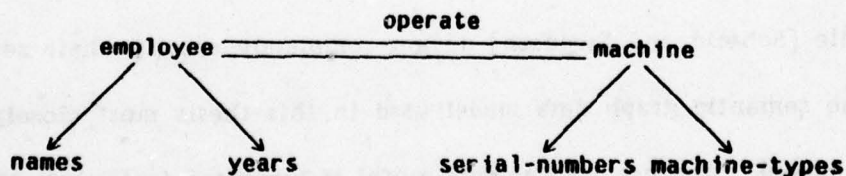


Figure 3.7 - Semantic Network Schema

The operations allowed in this data model are the insertion of independent

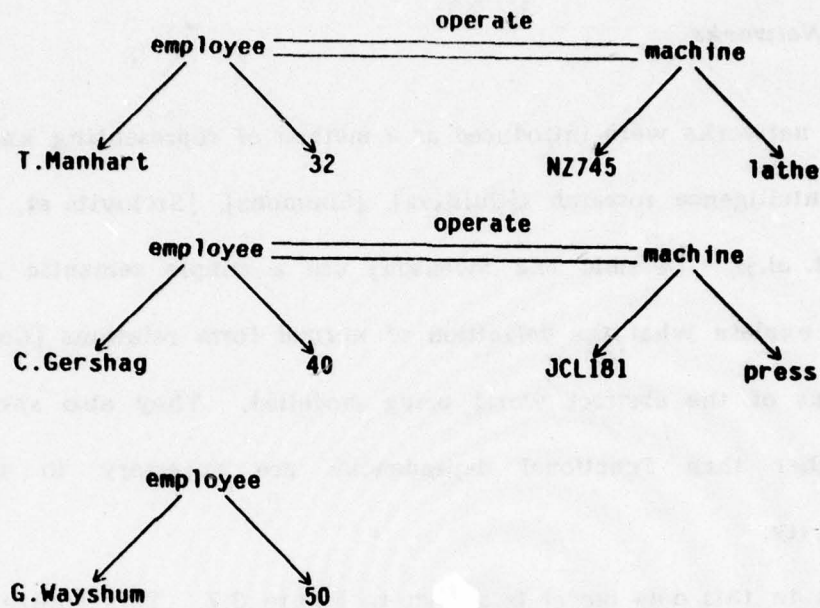


Figure 3.8 - Semantic Network State

object types along with their associated characteristic objects, the insertion of associations between two existing independent object types, the deletion of an association, and the deletion of an independent object so long as it is not in any associations. The data model includes the concept of key characteristics to identify objects.

While [Schmid and Swenson] do not completely specify their semantic data model, the semantic graph data model used in this thesis most closely resembles this data model. Another similar data model is presented in [Deheneffe et. al.].

3.4 The Entity-Relationship Model

[Chen] proposes a "unified" data model which has some features of a semantic network data model, yet uses a relational table to display the database state. The data model schema is partially specified by a diagram as in Figure 3.9.

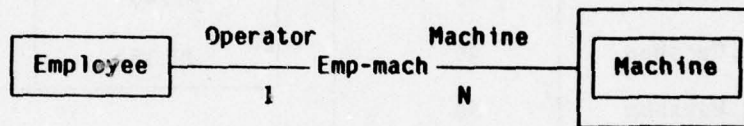


Figure 3.9 - Entity-Relationship Diagram

Figure 3.9 specifies two *entity sets*, Employee and Machine, which take on roles Operator and Machine in the *relationship* Emp-mach. This figure specifies a "1:N" relationship - i.e. a machine can only be in one Emp-mach relationship while an employee can be in any number of Emp-mach relationships. The double box denotes that the Machine entity set is a *weak* entity set and every machine must be in an Emp-mach relationship.

A database state is displayed in tabular form as in Figure 3.10. One relation is defined for each entity set with each tuple corresponding to an entity in the abstract world state. These *entity relations* define values for each attribute of an entity. A value set for each attribute specifies the possible

values of the attribute. Some group of an entity's attributes are specified as the *primary key* of the entity relation. The values of the primary key must uniquely specify the entity.

Employee		Machine	
name	age	number	type
names	years	serial-numbers	machine-types
T.Manhart	32	NZ745	lathe
C.Gershag	40	JCL181	press
G.Wayshum	50		

Emp-mach	
Employee	Machine
Operator	Machine
name	number
names	serial-numbers
T.Manhart	NZ745
C.Gershag	JCL181

Figure 3.10 - Entity-Relationship Relations

For each relationship there is a *relationship relation*, such as Emp-mach in Figure 3.10. A relationship relation specifies the primary key values for each participating entity in a relationship. Each tuple corresponds to a single

relationship in the abstract world state.

The operations in this data model, though not fully specified, are similar to operations in the semantic graph data model. Inserting an entity or a relationship corresponds to inserting a tuple in the appropriate entity relation or relationship relation. Deleting a relationship relation tuple causes automatic deletion of weak entities in the relationship. All operations must be in accordance with the stated relationship functionalities and entity keys.

Since each relational tuple corresponds to a single object in the abstract world state, this data model corresponds much more closely to the semantic graph model than the semantic relation data model of this thesis. In contrast to the entity-relationship data model which has very strict requirements on the allowed relations (i.e. one per entity set and one per relationship set), the semantic relation data model allows a large variety of relations to be defined for a single abstract world model.

3.5 Multiple Data Model Database System Architecture

The development of many different data models has both motivated and been motivated by the [ANSI] report which recommends a database system architecture based on the concept of several levels of database descriptions. This architecture allows for multiple users accessing a shared database. The three database description levels, shown in Figure 3.11, include the *internal schema*, the *conceptual schema* and *external schemas*.

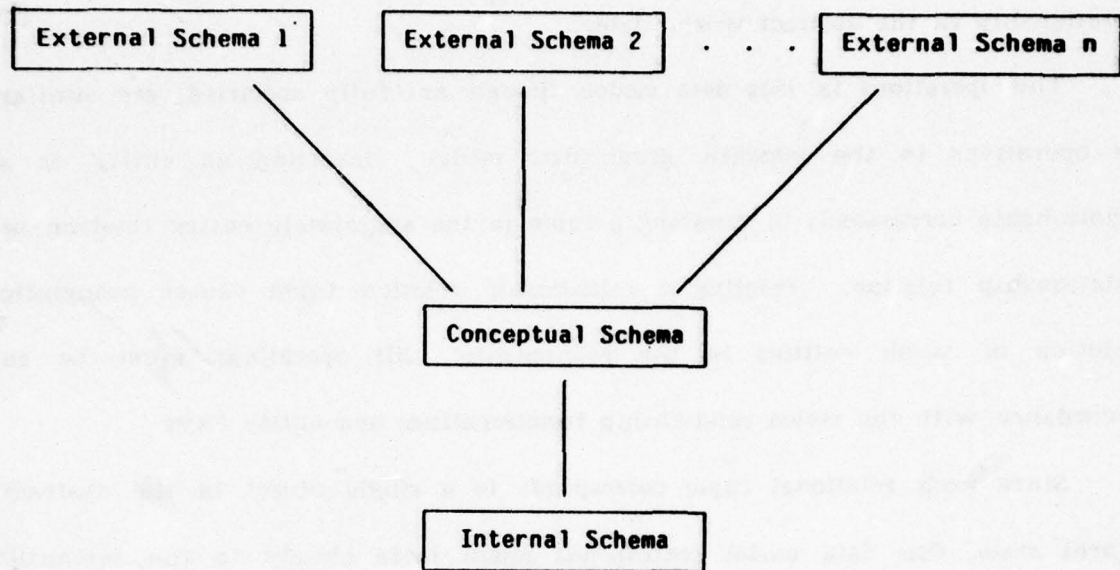


Figure 3.11 - ANSI Architecture

The internal schema specifies the types of data structures, devices and access methods which constitute the physical storage aspects of the database system. The conceptual schema specifies the contents of the database in terms of objects significant to the application being modelled in the database. There is much discussion over the exact form which this model should take. Suggestions, many of which we would classify as semantic data models, are found in [Nijssen 76]. Other appropriate data models are the data models defined in this thesis.

The separation of the conceptual schema from the internal schema results in data independence ([Date and Hopewell]). That is, the details of the physical storage representation can be altered without affecting the view of the database

presented to the user. The centralized conceptual schema guarantees, so long as only "appropriate" external schemas are allowed, that every user sees the database and performs updates consistent with the application being modelled.

Finally, external schemas provide the database system user with a view of the database appropriate to his specific needs and *desired data model*. The schemas are to be compatible with the particular programming or query language of the user. The external schema may present to the user just a subset of the information described in the conceptual schema. While this thesis is directly relevant to the case in which the external schema describes all of the data present in the conceptual schema, the definitions to be presented can be extended to handle the case where the external schema describes a subset of the conceptual schema.

This architecture requires several mapping functions - from the conceptual schema to the internal schema and from the conceptual schema to each external schema. For example, a query from a user posed in terms of an external schema must be translated into a query in terms of the conceptual schema, which must in turn be translated into a query which actually accesses the stored data. Results of the query must likewise be translated from an internal representation to a form appropriate to the user and the external schema.

The problem of how to define the mappings between schema levels so that all users see "equivalent" models of the application for *both* retrieval and update is extremely difficult for any case other than letting the external schema be a simple subset of the conceptual schema. [Klug and Tsichritzis] provide an

example of the complexities involved.

3.6 *Previous Work on Equivalence*

The difficulty of defining equivalent database states for syntactic data models can be seen by examining previous work on equivalence problems for the relational and DBTG (or similar) models. [Zimmerman] and [Fleck] both require that there be a relational tuple for each DBTG record plus a binary relational tuple for each DBTG set ownership-membership link. These restrictions on the form of the relational state, and hence schema, severely limit the types of information which a user might desire to appear together in a single relation.

[Kay] allows more general relations, but allows updates to be performed only on those relations whose tuples are in a 1-1 correspondence with the DBTG records and links. The earlier works of [Neuhold] and [McGee] allow much more general "equivalence" mappings, but ignore the problems of performing equivalent updates. [Klug and Tsichritzis] and [Paolini and Pelagatti] who also allow general mappings point out the difficulties of performing equivalent updates for arbitrary conceptual to external database state mappings.

[Navathe and Schkolnick] define a data model to facilitate "view integration". View integration is the process of developing a single model of the application consistent with each user's view of the application. This is similar to the problem of ensuring that each ANSI external schema is compatible

with the conceptual schema. Navathe and Schkolnick properly emphasize the importance of stating and taking into account the effects of operations altering the database state.

Chapter 4 - Framework for Data Model Definition

This chapter defines a framework for data model definition. By this, we mean that we will identify those components which constitute a formal data model definition. This framework will then be used in following chapters to formally define the semantic relation and semantic graph data models. All of the non-inferential data models discussed in Chapter 3 can also be defined in this framework.

A basic premise of what follows is that the database system is used to model some portion of the "real world" which is of interest to the system user. We will call this portion of the real world the *application*. Example applications are: a factory's inventory control system, the personnel assignments in a machine shop or an all-encompassing "corporate database". The *application state* represents a "snapshot" of the application at a given time.

The database state may consist of objects which are in a 1-1 correspondence with the application state (a common interpretation of the network approach) or it may consist of "statements" about the application state (an interpretation of the relational approach). Other approaches are possible. In any case, we assume that the database models some application which can be thought of as having a *state* and certain allowed *transitions* between states. In defining equivalences in this thesis we will be concerned about different data models' capabilities in expressing these application models, their states and their transitions.

A *data model* is defined as a (finite or infinite) set of *application models*.

$$\text{data model} = \{\text{application model}_1, \text{application model}_2, \dots, \text{application model}_n\}$$

Each of the application models specifies a view of an application to the database system.

4.1 Application Models

An application model does *not* specify the state of an application at a given point in time (with the exception of the "initial state"). This is the purpose of the database state. Rather, the application model specifies which database states are valid representations of application states and which transitions between states are allowed.

To specify the allowed states and transitions for each application, an application model consists of a *schema*, an *initial state* and a function which maps a finite set of *operation type names* into *operation types*. An operation type is the functional definition of the named operation.

$$\text{application model} = (\text{schema, initial state,} \\ \text{[operation type name} \rightarrow \text{operation type]})$$

The *schema* can be considered to be the "declarative" portion of the application model. A basic notion is that objects in the database state are *instances* of objects in the database schema. For example, corresponding to an *entity set* node in a semantic graph schema there are entity nodes in the

semantic graph state. Corresponding to a relational heading in a semantic relation schema there are tuples (statements) in the semantic relation state which "fit the form" of the relational heading.

Every time a user performs some operation on the database, the schema is consulted to make sure the operation is "legal". Consequently, the schema also contains constraints concerning the legality of applying operations to given states. In the semantic graph schema, the constraints take the form of the function defining arrowheads and in the distinction between solid and dotted edges. In the semantic relation schema, there is an explicit set of integrity constraints.

4.2 Operation Types

Operation types are the "procedural" portions of the application model. They specify what a user may do to change the database state. In the semantic relation data model, the operation types are: Insert-tuples and Delete-tuples which are used to insert or delete sets of tuples (statements). Some of the operation types for the semantic graph data model are: Insert-ind-entity for inserting an independent entity and Insert-semantic-unit for inserting a semantic unit.

For the data models studied in this thesis, all application models in a given data model have identical sets of operation types. This is also true of all data models in common use. However, it may be desirable to have data models which allow user-defined operation types. This bears relationships to the notion

of abstract data types [Liskov et. al.]. The desirability of having data models which allow user-defined operation types is discussed in the concluding chapter of this thesis.

Formally, each operation type is specified as a function

operation type : (schema \times arguments \times database state) \rightarrow database state.

Given a schema, a database state and some arguments whose types depend on the specific operation type, an operation type defines a new database state. One such possible new state is the *error* state.

For the study of formal equivalence properties, it is necessary to distinguish between *operation types* and *operations*. An *operation* is a function which maps each possible database state into another database state.

operation : database state \rightarrow database state

Given a schema and the set of possible other arguments for each operation type, we can generate an application model's *set of allowable operations*. For example, given a relational schema, there would be an operation corresponding to the insertion or deletion of each possible set of tuples.

The *initial state* of an application model specifies the database state which corresponds to the application state at "system initialization" time. This might, for example, be the "empty state". This state is of particular importance in that the closure of an application model's set of allowable operations applied to this initial state defines the set of *valid (reachable) states* for that application

model. When studying application model equivalence, we need only be concerned with those states which are actually reachable.

4.3 Databases

Thus far we have used the term *database state* without any further explanation. The specific form of the database state depends on the data model under discussion. In the semantic relation model, the state is formally expressed as a mapping from relation names to sets of statements. A semantic graph state is formally expressed as sets of nodes and edges.

Finally, we can define a *database* as specifying for an application both the current state and all possible state transitions.

$$\text{database} = (\text{application model}, \text{database state})$$

That is, in the context of a database system there is a static component, the application model, and a dynamic component, the database state. The application model determines the set of valid operations while the database state is the database system's representation of the application state.

There are several aspects of a database system which we will not explicitly explore in the framework or elsewhere in this thesis. However, the framework can easily be extended to discuss these issues. These issues are the questions of application model modification, concurrent access to a shared database and retrieval operations.

At a higher level there must be a means of declaring and perhaps modifying the application model. We will not explicitly discuss how this is to be done.

Similarly, a higher level view is needed to discuss the problems of concurrent access to shared databases. We will simply assume that to each user, it appears as though he were the only user of the system. Alternatives to this view are discussed in [Gray et. al.] and [Lamport].

We will not specify retrieval operations and query languages. Since we are assuming the database system to be non-inferential, retrieval operations can be thought of as simply displaying some subset of the database state.

The definitions used to define *data model* are summarized in Figure 4.1.

$\text{data model} = \{\text{application model}_1, \text{application model}_2, \dots, \text{application model}_n\}$
 $\text{application model} = (\text{schema}, \text{initial state},$
 $\quad \quad \quad [\text{operation type name} \rightarrow \text{operation type}])$
 $\text{operation type} : (\text{schema} \times \text{arguments} \times \text{database state}) \rightarrow \text{database state}$
 $\text{operation} : \text{database state} \rightarrow \text{database state}$
 $\text{database} = (\text{application model}, \text{database state})$

Figure 4.1 - Summary of Terms

Chapter 5 - The Semantic Graph Data Model

The fundamental concepts of the semantic graph data model are that the database state contains an object corresponding to each object in the application state, and that the operations applicable to the database state correspond to the "elementary" types of changes which can occur in the application state. By "elementary" we mean that these are the smallest units of change possible which would leave the database in a semantically consistent state. All other semantically meaningful state changes can be described as a sequence of elementary changes. We will describe the semantic graph data model in terms of the framework presented in Chapter 4.

5.1 The Semantic Graph Application Model Schema

The semantic graph application model schema describes the application by stating the general classifications of the objects to be modelled. It is assumed in the semantic graph model that each object can be classified, for example, as a characteristic, an entity or an association. Examples of these classifications were presented in Chapter 2. Since each type of database object has its own set of allowed operations, the formal basis for deciding what type of object should be used to model a real-world object is according to what kinds of state changes should be allowed on the database representation of the real-world object.

In set theoretic terms, a semantic graph schema is

$$(E, A, D, r, c, fr, fc, sd, R, C, Str)$$

where the components are defined as follows:

E: This is a finite set of *entity set names*. Each element of this set corresponds to a unique *entity set node* in the graph representation of the schema. Examples of elements of this set, drawn from the machine shop example presented in Chapter 2, are "employee" and "machine".

A: This is a finite set of *association set names*. Each element of this set corresponds to a unique *association set node* in the graph representation of the schema. An example of an element of this set is "operation".

D: This is a finite set of *domain names*. Each element of this set corresponds to a *domain node* in the graph representation. Examples are "names", "years", "serial-numbers" and "machine-types".

r: This is the *role set edge function*. *r* is a function

$$r: (A \times R) \rightarrow E$$

where *R* is the set of *role names*. Role names are such things as "agent" and "object" which are derived from the idea of cases in case grammars as mentioned in Chapter 2. The roles we use in examples are based on the work of [Fillmore]. Each role set edge is shown in the graph representation as a line connecting an association set node and an entity set node. The line is labelled

with the role name. The direction of the edge will explained later in this chapter. r is a function because we require that a given role name may label only a single edge terminating on any particular association set node. This is simply a statement that a role in an association is only fulfilled by one type of entity.

c : This is the *characteristic set edge function*. c is a function

$$c: (EUA \times C) \rightarrow D$$

where C is the set of *characteristics* such as "name", "age", "number" and "type". Each characteristic set edge is shown in the graph representation as a line connecting an entity set node or an association set node with a domain node. The line is labelled with the characteristic name. The direction of the edge will be explained later in this chapter. A given characteristic name may label only one edge terminating on any particular entity set or association set node.

The functions fr and fc will specify restrictions on the values of characteristics and on the combinations of entities into associations which may occur in the database state. These restrictions serve two purposes. Firstly, they express restrictions which occur in the application such as: "Each machine may have only one operator." Secondly, they provide means by which objects in the database state can be identified. That is, "A machine is uniquely identified by its serial number."

fr: This is a function

$fr: (A \times R) \rightarrow (\{restricted, unrestricted\} \times \{mandatory, non-mandatory\})$.

This function determines two *features* of role set edges. A role set edge may be mandatory or non-mandatory. An edge designated mandatory means that every entity in the database state of the type connected to the edge must be a participant in at least one association of the type also connected to the edge. In the machine shop example, the edge connecting the machine entity set node with the operation association set node is such an edge because we require that each machine must be operated. The feature which is specified as restricted or unrestricted will be discussed in the definition of *fc* below.

In the graph representation of the schema, we will show mandatory edges as solid lines while non-mandatory edges will be shown as dotted lines. An edge which is unrestricted will have an arrowhead terminating on the entity set node while a restricted edge will have an arrowhead terminating on the association set node.

fc: This is a function $fc: (EUA \times C) \rightarrow \{key, non-key\}$. It assigns a feature to each characteristic set edge. The significance of this in the case of characteristic set edges connected to entity set nodes is that in the database state there may be only a single entity node connected to a given set of domain element nodes if all of the corresponding characteristic set edges have the key feature. In other words, if an entity set node has at least one characteristic set edge with the key feature, then in the database state, each entity of that type

can be uniquely identified by its key characteristics. An example is the "name" characteristic being key for the "employee" entity set and the "number" characteristic being key for the "machine" entity set.

The significance of the feature for association nodes which have characteristics is somewhat different and involves the restricted or unrestricted feature specified by the function *fr*. Associations in the database state are uniquely identified by key characteristics of an association along with the key characteristics of entities in restricted roles of the association.

If an association set has no key characteristics, then the restricted entities uniquely identify the association in the database state. If an association set has no restricted roles, then the key characteristics uniquely identify the association. For an association which has both restricted roles and key characteristics, then it is a combination of these roles and characteristics which can uniquely identify an association of this type in the database state. An example of an association set with key characteristics will be presented later in this chapter.

If a characteristic set edge has the key feature, then an arrowhead is placed on the end of the edge terminating on the entity or association set node. If the feature is non-key, then an arrowhead is placed on the domain node end of the edge.

sd: This is a function which provides the definition of domain element representation. Domains in the application are sets of semantic concepts such as

"the number 1" or "the color red". In the set theoretic formalisms we are giving which are to model these concepts we must eventually get to the point where we agree on symbolic representations for these concepts. The function sd specifies such a symbolic representation of each of the domains named in the set D .

We will require that every domain element can be expressed as a character string, though not necessarily a single word. We will also require that for a given domain, distinct character strings represent distinct abstract domain elements. Note that this is not true of natural language. Both "4" and "IV" represent the same abstract number. A domain may be infinite such as the domain "natural numbers" or a domain may be finite such as the domain "sex" with elements "male" and "female".

sd will then be a function

$$sd:D \rightarrow [Str \rightarrow \{true, false\}].$$

The function maps each element of D into a *domain specification*. A domain specification is a function which maps each possible character string into true or false indicating whether or not the character string is a valid representation of some domain element. We will not further specify the set Str of possible character strings. A set appropriate to the application being modelled is assumed.

We have seen above that the features specified by the functions fr and fc restrict the possible combinations in the database state of entities into

associations and of characteristics of entities and associations. This provides a means of identifying entities by their key characteristics and of identifying specific associations by specifying the identities of the entities participating in the restricted roles of the association and the key characteristics of the association.

We are now going to make the assumption that each entity type has at least one key characteristic and that each association type has at least one key characteristic or at least one restricted role. This eliminates the possibility of having two entities of the same type with identical characteristics. It also eliminates the possibility of having two associations of the same type which have both identical characteristics and entities participating in each role which have identical characteristics.

The sets *R* and *C* have been described in connection with *r* and *c*. *Str* has been described in connection with *sd*.

This concludes the set definitions of the semantic graph schema. Figure 5.1 shows a more complicated schema which will be used in subsequent discussions. Some features of this schema are:

1. The entity set *machine* now has key characteristics, *type* and *number*, and non-key characteristics, *age* and *location*.
2. The *operation* association now has a key characteristic, *shift*, and a non-key characteristic, *cost-per-hour*. That is, both a *shift* and a *machine* identification are needed to identify a particular operation association. And the *cost-per-hour* to operate a machine is a characteristic of a particular machine on a particular shift.

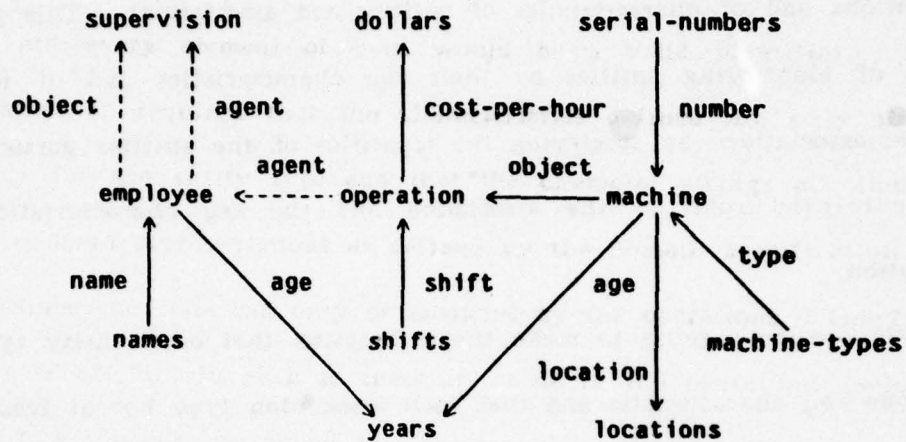


Figure 5.1 - More Complex Semantic Graph Schema

3. The *supervision* association in this schema has both the *agent* and *object* roles from the entity set *employee*.
4. The domain *years* is used by characteristics of both the *employee* and the *machine* entity sets.

5.2 The Semantic Graph Database State

Before defining the operation types which constitute the second component of an application model, we will give the formal definition of the database state on which the operations are to be used. The database state is a model of the application state. For each object in the application state, there is a corresponding object in the semantic graph database state. Every database state is based on some schema. For the following definitions, we will assume that an appropriate schema

(E, A, D, r, c, fr, fc, sd, R, C, Str)

has been defined. A semantic graph database state is then represented in set theoretic terms as

$$(E, A, D, r, c)$$

where the components are defined as follows:

D: This is the set of *domain element nodes*. Each element in this set is an element of $(D \times \text{Str})$. Each node specifies a domain name and the character string representation of some element in the domain.

Example elements of *D* are (names, T.Manhart) and (serial-numbers, NZ745). In the graph representation of the state we will show only the domain element character string as the node.

E: This is the set of *entity node identifiers*. Each element of this set corresponds to an entity node in the graph representation of the database state. The identifier consists of the values of those characteristics which uniquely identify the node. Each entity node identifier in *E* is an element of the set *E-id* defined as

$$E\text{-id} = (E \times (C \rightarrow (D \times \text{Str}))).$$

Each identifier specifies an entity set name giving the type of the entity and a partial function which maps each key characteristic of the entity into the domain type and string which represents the domain element for that characteristic. An example would be (employee, {(name, (names, T.Manhart))}). Note that we use a set of ordered pairs to specify a function. This will be

done throughout this thesis.

Clearly, not every element of $E\text{-id}$ would be a valid identifier. For example, it must be required that the characteristics defined are only those which are key for the entity type and that the character strings are indeed valid domain element representations as defined by the domain specification in the schema. Since the state can only be modified by the operations defined in the application model, checks such as these are made in the operations (which will be defined in the next section of this chapter).

The elements of E , and the corresponding nodes in the graph, can be partitioned into *entity sets* according to the entity set name which is the first component of the identifier. In the graph representation of the state, the entity set name is used as the node label, but this labelling is not necessarily unique. However, there will be a unique node which has the label specified by the identifier and has edges connecting the node to the characteristic domain elements specified. That is, there are two "machine" nodes in Figure 2.2, but they are distinguished by the "number" characteristic.

A: This is the set of *association node identifiers*. Each identifier is an element of the set $A\text{-id}$ defined as

$$A\text{-id} = (A \times (R \rightarrow E\text{-id}) \times (C \rightarrow (D \times \text{Str}))).$$

An association identifier must specify the association set name, the entity identifiers of each entity in a restricted role, and the domain element for each key characteristic. As with entity sets, the elements of A are partitioned into

association sets by the specified association set names. The update operations must enforce certain requirements such as that of making sure that each entity identifier has an entity set name which is the same as that which the schema specifies should fill this role. Such constraints are enforced (and defined) by the operation type definitions.

An example element of A would be:

(operation, ((object, (machine, ((number, (serial-numbers, NZ745)))))), ()).

r : This is the set of *role edges*. Role edges connect entity nodes with association nodes specifying the role which the entity has in the association. r is a subset of

$$(A-id \times R \times E-id).$$

The type of the entity node for each role must be the same as that specified by the role set edges in the schema. It is required that every role defined for an association set in the schema has a corresponding role edge for each association in the database state. Since each association may have only one entity for each role, r will be expressed as a partial function

$$r: (A-id \times R) \rightarrow E-id.$$

In the graph representation of the state, these edges connect the association specified to the entity specified with a line labelled by the role name.

c : This is the set of *characteristic edges*. It is a subset of

$$(E-id \cup A-id \times C \times (D \times Str)).$$

Each element specifies an entity or association node and the domain element

node which is the specified characteristic. Each element of E and A must be connected to an edge corresponding to each of the characteristic set edges specified in the schema for that type of entity or association. c will be expressed as a partial function

$$c: (E-idUA-id \times C) \rightarrow (D \times Str).$$

This completes the set definitions of the database state. To fully specify all of the constraints on the elements in these set, we must look at the operation types to be defined in the next section which modify the state. Figure 5.2 shows a database state in graph notation corresponding to the schema of Figure 5.1.

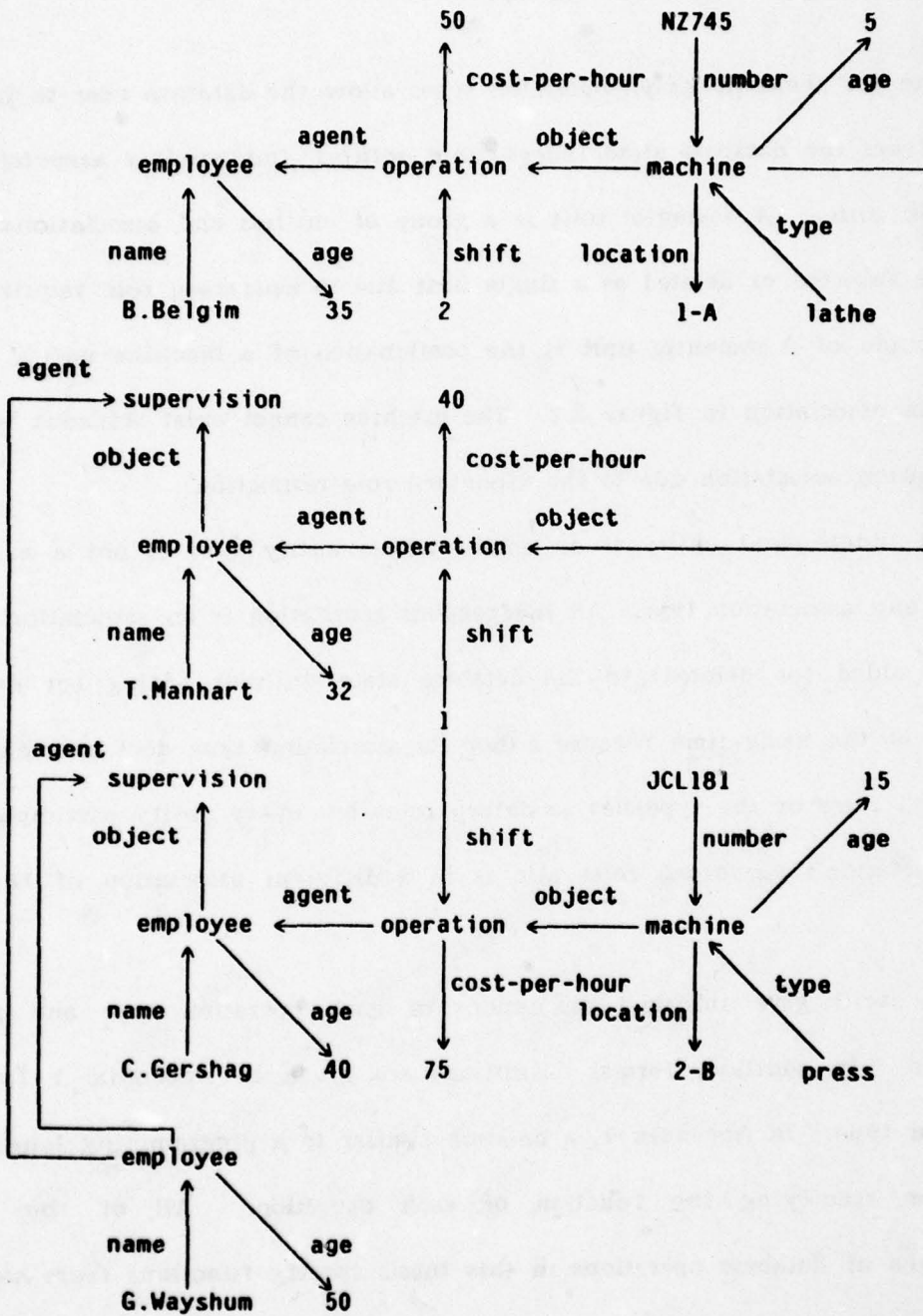


Figure 6.2 - More Complex Semantic Graph State

5.3 The Semantic Graph Operation Types

The six semantic graph operation types allow the database user to insert or delete from the database state *independent entities*, *independent associations* or *semantic units*. A semantic unit is a group of entities and associations which must be inserted or deleted as a single unit due to mandatory role requirements. An example of a semantic unit is the combination of a machine entity and an operation association in Figure 2.2. The machine cannot exist without being in an operation association due to the mandatory role restriction.

An independent entity is an entity whose entity type is not a mandatory role in any association type. An independent association is an association which can be added (or deleted) to the database state without adding (or deleting) entities at the same time because either its association type does not have any mandatory roles or the type has mandatory roles but every entity participating in the association's mandatory roles also is in a different association of the same type.

We will give informal discussions of each operation type and provide examples. In addition, formal definitions are given in Appendix 1 for each operation type. In Appendix 1, a notation similar to a programming language is used for specifying the function of each operation. All of the formal definitions of database operations in this thesis specify functions from some set of arguments to the set of database states. The definitions do not specify a specific computer implementation of the data models discussed. The basic

constructs of the notation are similar to the applicative language of [Dennis et. al.] and could be equivalently represented in lambda calculus notation [McCarthy]. The various type domains are specified in Figure A1.1 of Appendix 1. A Pascal-like notation [Wirth] is used to select elements from tuples (i.e. like Pascal records). We will not specify the semantics of the notation further.

Figure A1.2 specifies certain functions which are regarded as predefined. Several of these are based on an assumed ordering of the sets of role names and association type names. We will also assume there is a special value error in every type domain and that any function which is given error as an argument will evaluate to error.

5.3.1 Insert-ind-entity

Figure A1.5 defines the operation type which is used to add independent entities to the database state. The function Insert-ind-entity maps a graph schema, a graph state and an entity description into a graph state. The auxiliary function valid-entity defined in Figure A1.3 is used to check that the entity description is in proper form and that there does not already exist an entity in the database state with the same key characteristics.

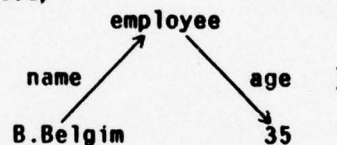
If these tests are passed and if the entity type is truly independent, i.e. it is not a mandatory role for any association type, then Insert-ind-entity evaluates to a database state which is identical to the argument state except for: i). the addition of the entity id to the set of entity node identifiers, ii). the addition to

the set of domain element nodes those domain elements used to define the entity characteristics, and iii). the addition of those characteristic edges which connect the entity node and the domain elements.

For the reader's convenience, Figures 2.1 and 2.2 are repeated as Figures 5.3 and 5.4. These figures will be used in demonstrating operations in this chapter. All example operations will be specified using both a graphical notation and a formal set notation which is appropriate to the definitions in Appendix 1.

Figure 5.5 shows the database state resulting from

Insert-ind-entity(schema5.3, state5.4, (5-1)



or in set notation

Insert-ind-entity(schema5.3, state5.4, (5-1)
 ((employee, {(name, B.Belgim)}), {(name, B.Belgim), (age, 35)}))

where schema5.3 represents the schema of Figure 5.3 and state5.4 represents the state of Figure 5.4. Domain names are left out for brevity. As previously mentioned, functions are denoted in operation specifications by a set of ordered pairs.

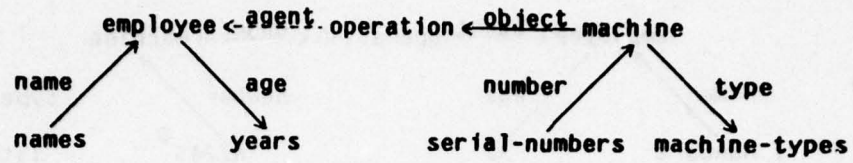


Figure 5.3 - Semantic Graph Schema (same as Figure 2.1)

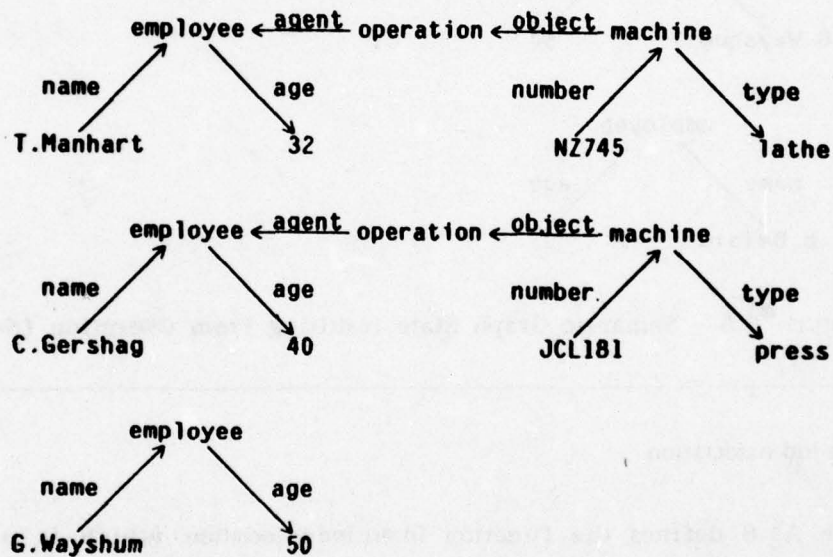


Figure 5.4 - Semantic Graph State (same as Figure 2.2)

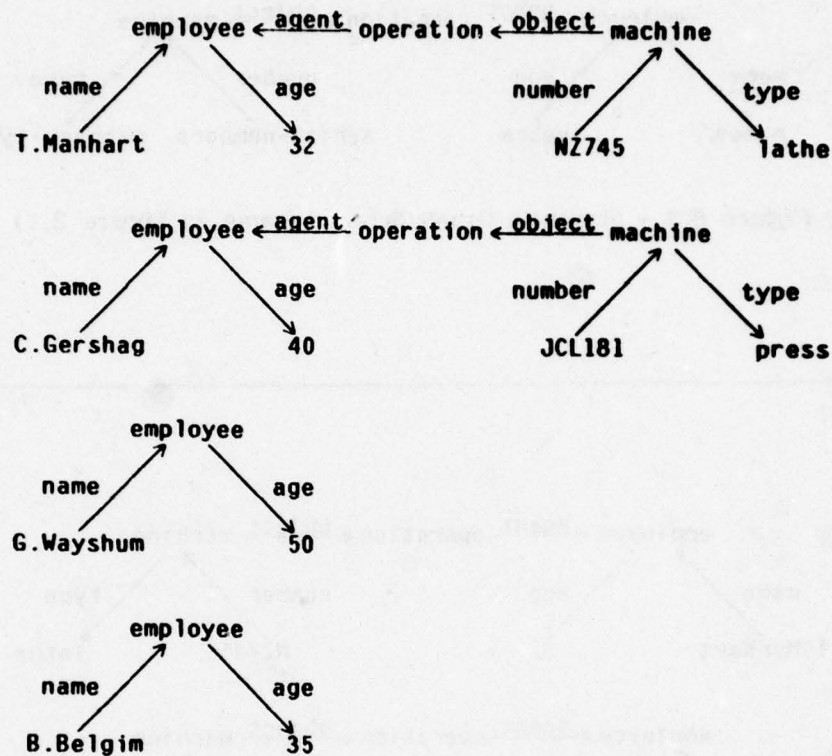


Figure 5.5 - Semantic Graph State resulting from Operation (5-1)

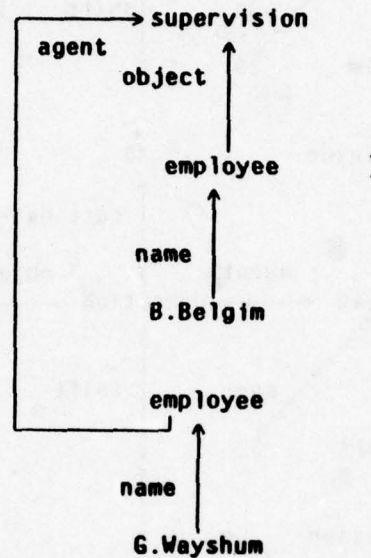
5.3.2 Insert-ind-association

Figure A1.6 defines the function Insert-ind-association which is used to add independent associations to a semantic graph state. It uses the function valid-association defined in Figure A1.4 to make sure that the association description is of proper form and that an association with identical key characteristics and restricted roles is not already in the database state. In addition to these tests, the other basic requirement is that all of the entities which participate in the association must already be in the database state. If all

of these requirements are met, then the state is returned with appropriately inserted association nodes, domain nodes, role edges and characteristic edges.

Figure 5.6 shows the database state resulting from

Insert-ind-association(schema5.1, state5.2, (5-2)



or in set notation

Insert-ind-association(schema5.1, state5.2, (5-2)

```

((supervision, {(agent, (employee, {(name, G.Wayshum)}))},
  (object, (employee, {(name, B.Belgim)}))}, {}),
  {(agent, (employee, {(name, G.Wayshum)}))}
  (object, (employee, {(name, B.Belgim)}))}, {}))
  
```

where schema5.1 is the schema of Figure 5.1 and state5.2 is the state of Figure 5.2.

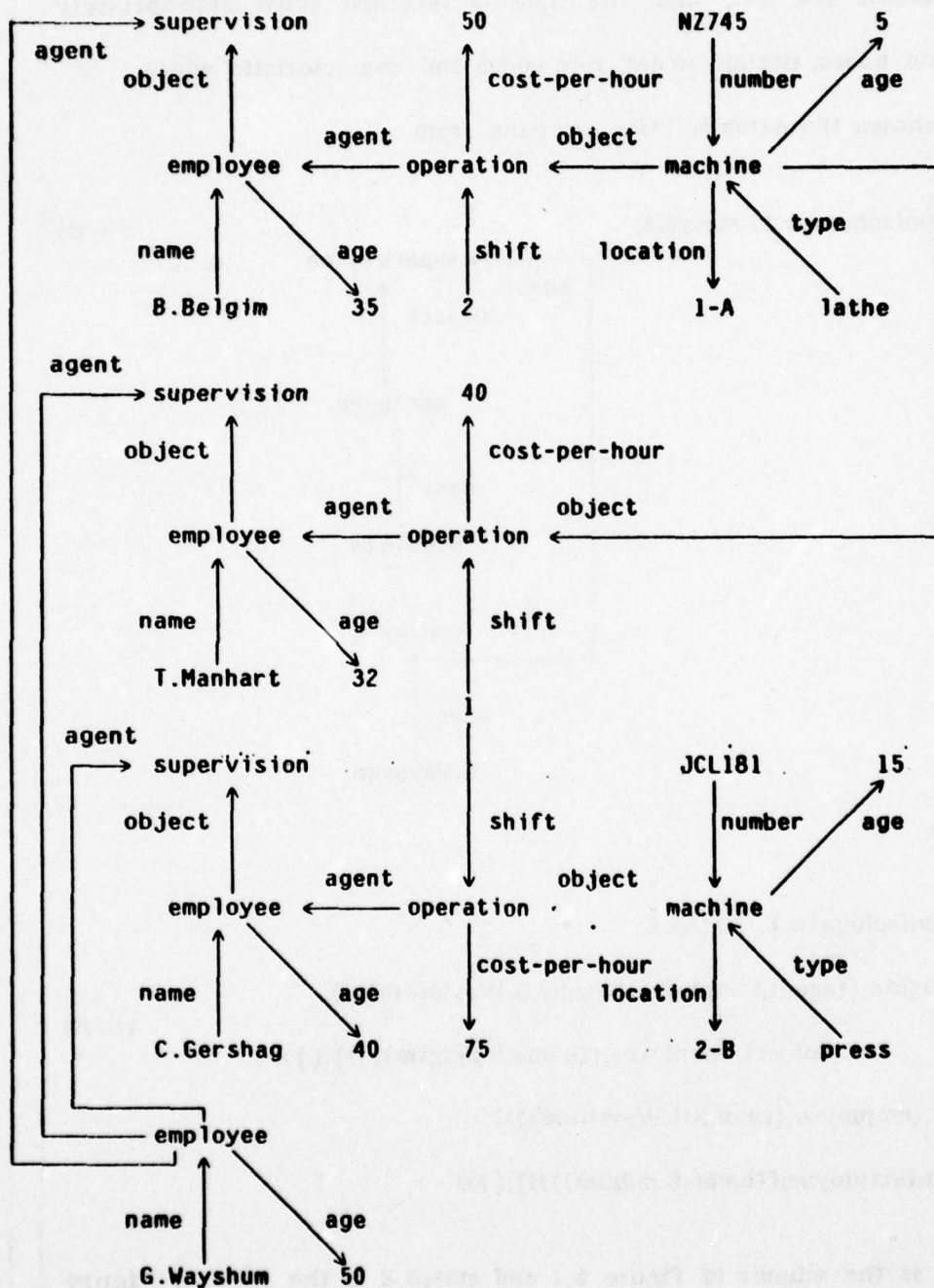


Figure 5.6 - Semantic Graph State resulting from Operations (5-2)

5.3.3 Insert-semantic-unit

The functional definitions for operations which insert semantic units are more complicated than those discussed so far. Figures A1.7, A1.8, A1.9 and A1.10 define the necessary functions. The function *Insert-semantic-unit* corresponds to the function which would be visible to a database system user. This function requires as arguments a schema, a state and an *arg-list*.

The *arg-list* consists of entity descriptions and association descriptions for all of the components of the semantic unit. The way in which these descriptions are ordered in the list is not significant in terms of the application. However, an ordering is chosen for the purpose of defining the operation type functions.

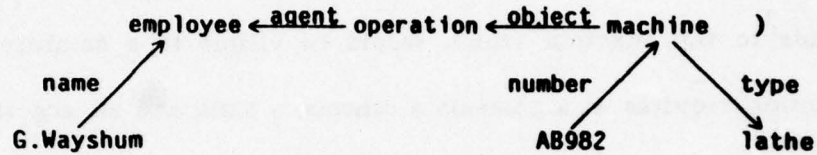
The first element in the *arg-list* is an arbitrarily chosen entity in the semantic unit. From this starting point, the ordering of the arguments corresponds to a depth first traversal of portions of the schema along role set edges beginning from the entity set node corresponding to the first entity in the list. For each association set node, all edges are traversed. For each entity set node or association set node which is visited, an appropriate entity or association description is added to the *arg-list* unless the description has been added earlier in the list or if the particular entity or association already exists in the database state.

The function *Insert-semantic-unit* initiates the mutually recursive calls of the functions *insert-mand-entity*, *insert-mand-association* and *check-roles* which traverse the

schema and do the insertions. The functions valid-entity and valid-association are also used to make sure that the descriptions are acceptable.

Figure 5.7 shows the database state resulting from

Insert-semantic-unit(schema5.3, state5.4, (5-3)



or in set notation

Insert-semantic-unit(schema5.3, state5.4, (5-3)

```

((machine, {(number, AB982)}), {(number, AB982), (type, lathe)}),
((operation, {(object, (machine, {(number, AB982)}))}, {}),
{(agent, (employee, {(name, G.Wayshum)}))},
(object, (machine, {(number, AB982)}))}, {})))).
  
```

In this case, the traversal of the schema (Figure 5.3) starts at the machine entity set node. So the first element in the *arg-list* is the description of the machine entity being inserted. The first (and only) mandatory role set edge terminating on the employee entity set node is the object role of the operation association. So this edge is followed and the next element of the *arg-list* is the association description which includes the previously defined machine entity in the object case. Since all entities participating in this association are already either in the state or in the *arg-list* this concludes the traversal.

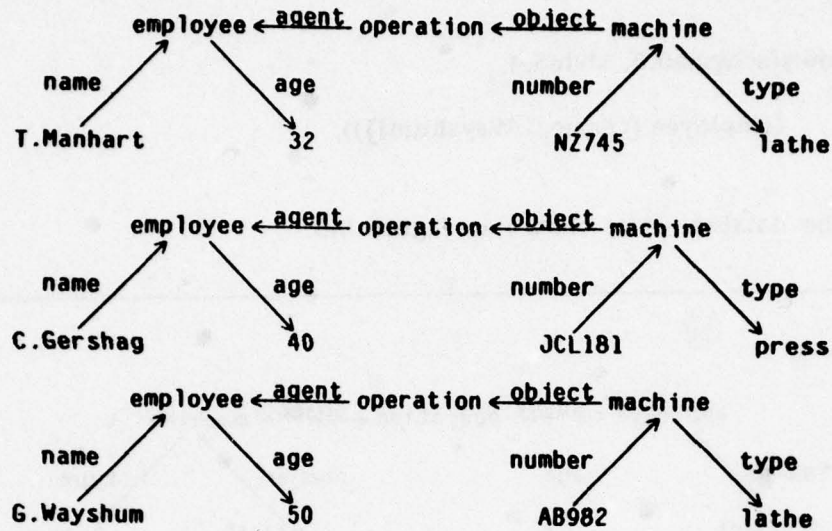


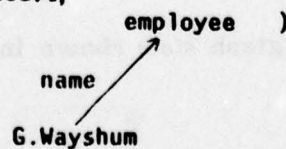
Figure 5.7 - Semantic Graph State resulting from Operation (5-3)

5.3.4 Delete-ind-entity

The operation type used to delete independent entities is defined by the function Delete-ind-entity in Figure A1.11. The basic requirement that must be fulfilled by the entity to be deleted is that it must not be in any associations. This requirement implies the independence of the entity.

The operation

Delete-ind-entity(schema5.3, state5.4, (5-4)



or in set notation

Delete-ind-entity(schema5.3, state5.4,

(employee, {(name, G.Wayshum)})),

(5-4)

results in the database state shown in Figure 5.8.

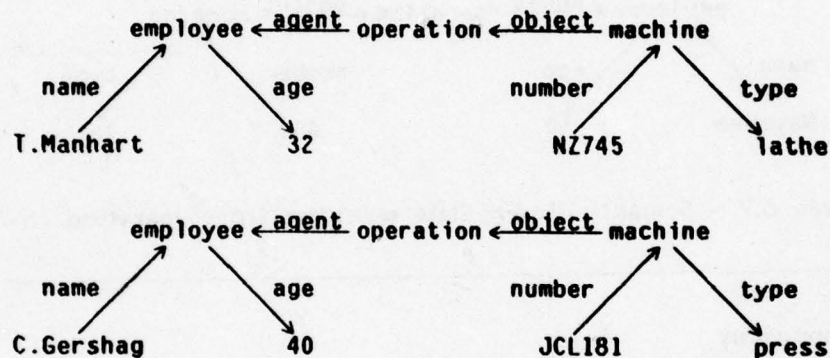


Figure 5.8 - Semantic Graph State resulting from Operation (5-4)

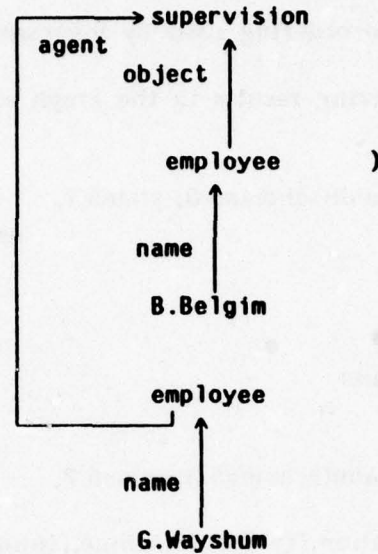
5.3.5 Delete-ind-association

The operations to delete independent associations are defined by the function Delete-ind-association in Figure A1.12. A check is made to make sure that this association is in fact independent.

If state5.6 represents the graph state shown in Figure 5.6, then

Delete-ind-association(schema5.1, state5.6,

(5-5)



or in set notation

Delete-ind-association(schema5.1, state5.6,

(5-5)

(supervision, {(agent, (employee, {(name, G.Wayshum)})),
(object, (employee, {(name, B.Belgim)}))})))

results in the state shown in Figure 5.2.

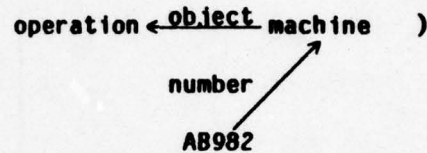
5.3.6 Delete-semantic-unit

To delete a semantic unit, the function Delete-semantic-unit with its auxiliary functions delete-roles and delete-mand-entity are used. These functions are defined in Figures A1.13, A1.14 and A1.15. In addition to the schema and state, the identification of any arbitrarily chosen association in the semantic unit are given as arguments to Delete-semantic-unit. The mutually recursive

functions then delete each entity or association in the semantic unit in the "reverse" of the ordering used by Insert-semantic-unit.

The following results in the graph state shown in Figure 5.4:

Delete-semantic-unit(schema5.3, state5.7, (5-6)



or in set notation

Delete-semantic-unit(schema5.3, state5.7, (5-6)
 (operation, {(object, (machine, {(number, AB982)}))}, {})).

This completes the presentation of the functions which define the operation types.

5.4 Definitions of Valid Operations and Valid States

Now we will draw the distinction between operation types and operations. The functions presented in Appendix 1 define the six operation types. Each operation type is a function which takes as arguments a semantic graph schema, a semantic graph state and one other argument depending on the operation type. However, we want an operation to be a function which maps one database state into another database state.

The set of *valid operations* will be necessary for the study of equivalence properties in later chapters of this thesis. We will first show how such a set of operations is defined for the Insert-ind-entity operation type. For a given choice of base finite sets E , A , D , R , C and Str , the set of possible entity descriptions, $E\text{-desc}$, is finite. Then define the set of possible independent entity insertion operations for a given semantic graph schema $g\text{-schema}_n$ as the set of functions

```
operationj(g-state:G-state)
  returns G-state

  return Insert-ind-entity(g-scheman,g-state,e-descj)
```

where $e\text{-desc}_j \in E\text{-desc}$.

The operations for the other operation types are defined similarly.

Definition 5.1: For a given $g\text{-schema}_n \in G\text{-schema}$, the set of valid operations is the union of the following sets for $1 \leq i \leq 6$:

```
{operationij(g-state:G-state)
  returns G-state

  return Op-typei(g-scheman,g-state,argj)}
```

where for:

- i = 1: Op-type_i = Insert-ind-entity
arg_j \in E-desc
- i = 2: Op-type_i = Insert-ind-association
arg_j \in A-desc
- i = 3: Op-type_i = Insert-semantic-unit
arg_j \in Arg-list
- i = 4: Op-type_i = Delete-ind-entity
arg_j \in E-id
- i = 5: Op-type_i = Delete-ind-association
arg_j \in A-id
- i = 6: Op-type_i = Delete-semantic-unit
arg_j \in A-id

Finally, we would like to define the set of *valid semantic graph states* for a given application model. These are those elements of the set G-state which represent possible states of the application. For purposes of this study, we will assume that the only allowed initial state is the empty state.

Definition 5.2: The set of valid semantic graph states for a given semantic graph schema g -schema and set of valid operations G -operations, where G -operations is defined for g -schema as in Definition 5.1, is defined as follows:

1. The empty state = $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ is a valid semantic graph state.
2. If g -state is a valid state, $g\text{-op} \in G\text{-operations}$ and $g\text{-op}(g\text{-state}) \neq \text{error}$, then $g\text{-state}' = g\text{-op}(g\text{-state})$ is a valid semantic graph state.

Chapter 6 - The Semantic Relation Data Model

As previously stated, the semantic relation data model is based on the concept that the user sees the database state as sets of statements about the application state. In describing the semantic relation data model we will make the assumption that the application and application state are formally specified by the corresponding semantic graph application model and semantic graph database state respectively.

6.1 Semantic Relation Statements

The meaning of a statement (i.e. row or tuple) in a relation can be given by a logical statement about the application state. Several examples will be given which will both clarify this idea and show the "expressive power" of the semantic relation data model. The user of a semantic relation database system would not deal explicitly with the logical statements as will be presented here. We will use "row", "statement" and "tuple" interchangeably.

For the examples to follow we will consider Figures 6.1, 6.2, 6.4 and 6.5 which are identical, with the exception of some italicized words, to Figures 2.3, 2.4, 2.6 and 2.7. The significance of the italicized words in the relations will be explained shortly.

Using the notation which defines the semantic graph data model as in Chapter 5 and Appendix 1, we will associate with the first row of the relation in Figure 6.1 the following logical statement:

```
(∃a-id:A-id) (∃e-id-1:E-id) (∃e-id-2:E-id)
(
  a-id ∈ state.A
  ∧ e-id-1 ∈ state.E
  ∧ e-id-2 ∈ state.E
  ∧ a-id.type = operation
  ∧ e-id-1.type = employee
  ∧ e-id-2.type = machine
  ∧ state.R(a-id, agent) = e-id-1
  ∧ state.R(a-id, object) = e-id-2
  ∧ state.C(e-id-1, name) = (names, T.Manhart)
  ∧ state.C(e-id-2, number) = (serial-numbers, NZ745)
  ∧ state.C(e-id-2, type) = (machine-types, lathe) ).
```

operate:agent	operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 6.1 - Semantic Relation

This asserts that there exist in the application state an association and two entities with the specified characteristics.

Note that a tuple concerning the "operate" predicate was translated into a logical statement concerning an "operation" association. This points out that the relation rows are to be thought of as natural language statements given in a case grammar form. In fact, we will assume that the set of possible cases is identical to the set of possible roles. For each association we will assume there is a unique corresponding predicate. However, there is one group of predicates which do not correspond to associations.

For each type of entity in the application, there is a predicate "be type" where type is the entity set name. An example of a relation using such a predicate is in Figure 6.2. This type of predicate is used to state the existence of an entity as opposed to the existence of an association. With the first row in Figure 6.2 we can associate the logical statement

```
( $\exists$ e-id:E-id)
(  e-id  $\in$  state.E
   $\wedge$  e-id.type = employee
   $\wedge$  state.C(e-id,name) = (names,T.Manhart)
   $\wedge$  state.C(e-id,age) = (years,32)).
```

Note that the above logical statement asserts the existence of an employee named "T.Manhart", but it does not state that there is only one such employee. This information would be derived from a relational constraint stating that the relation in Figure 6.2 is a function from the values in the first column to the values in the second column. That is, the names in the first column are unique. With such a constraint, using " $\exists!$ " to mean "there exists exactly one", we would state

be employee:object	
employee	
<i>name</i>	<i>age</i>
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Figure 6.2 - Semantic Relation

```
( $\exists!$  e-id:E-id)
(  e-id  $\in$  state.E
   $\wedge$  e-id.type = employee
   $\wedge$  state.C(e-id,name) = (names,T.Manhart)
   $\wedge$  state.C(e-id,age) = (years,32)).
```

This states that there is a unique employee named "T.Manhart" and of age "32". We might also add to the logical statement that "name" is the identifying (key) characteristic.

The information necessary to know that a relational statement concerns a unique entity or association is contained in the constraints. However, it is such a useful piece of information when interpreting the meaning of a relation, that we will introduce a "sugaring" to express the information in the relation itself. This is the purpose of the italicization in Figures 6.1 and 6.2.

For explanatory purposes we need the concept of a *column-group*. By

column-group we will mean the subset of a relation's columns which contains all of the characteristics described in the relation for a given predicate or set of predicate:case pairs. In Figure 6.1, the employee "*name*" column forms one column-group. The machine "*number*" and "*type*" columns form the second column-group of this relation.

If all of the key characteristics of an entity type are present in a single column-group, then the key characteristic names in the relational heading will be italicized. This is the reason for the italicized "*name*" and "*number*" in Figure 6.1. If some, but not all of the key characteristics of an entity type are present, none of the characteristic names for the column-group will be italicized. This will clearly delineate those cases in which unique entities are referenced.

A similar scheme will be adopted for the predicates in the relations. If, for a given predicate in a relation, all of the key characteristics are present and all of the key characteristics for each restricted case (role) are present, then the key characteristic names and restricted case names will be italicized. This accounts for the italicized "*object*" in Figure 6.1. If any of the identifying characteristics are missing, then none of the relevant case names or characteristic names will be italicized.

This italicization is not directly reflected in the formal specification of the relation headings. Such information is specified by the constraints. The reason for expressing this information in terms of the constraints is to maintain greater generality in the semantic relation data model.

While the italicization scheme is perfectly acceptable for describing applications as specified by semantic graph application models, we can imagine features of applications which, while not expressible in the semantic graph model, would be expressible in the semantic relation model with appropriate types of constraints. In terms of the problem of uniquely identifying entities and associations, we might, for example, desire multiple identifiers for a single entity or association. These problems of comparative expressiveness of the data models will be explored further in succeeding chapters of this thesis.

It is *very important* to realize that the use of italics in this thesis is not identical to the notion of relational keys in [Codd 72a]. In particular, the set of all italicized characteristics do not *necessarily* comprise a relational key. For example, Figure 6.3 shows a relation describing a state in which a unique employee name "J.Doe" operates two types of machines. The employee "*name*" column does *not* form a key to this relation. Also note that Figure 6.3 does not rule out the possibility that "J.Doe" operates several machines of type "drill".

The relational key of Codd provides a means of uniquely identifying relational tuples in the database state. Italicization (and constraints) in the semantic relation data model provides a means of uniquely identifying entities and associations in the application state. At times the goals of relational keys and italicization coincide. In fact, many of the benefits of relational normalization theory [Codd 72a] result from structuring relations so that single tuples correspond to single objects in the application state.

The use of relations to express a statement concerning a single association

operate:agent		operate:object
employee		machine
<i>name</i>	age	type
names	years	machine-types
J.Doe	25	drill
J.Doe	25	lathe

Figure 6.3 - Italicization Not the Same as Relational Keys

is the basic type of semantics associated with the use of relations in [Chen] and [Hall et. al.]. However, the model presented here is unique in its ability to express more complex statements. For example, the first row of the relation in Figure 6.4 is associated with the following logical statement:

supervise:agent	supervise:object operate:agent	operate:object
employee	employee	machine
<i>name</i>	<i>name</i>	<i>number</i>
names	names	serial-numbers
G.Wayshum	T.Manhart	NZ745
G.Wayshum	C.Gershag	JCL181

Figure 6.4 - Semantic Relation With Set of Predicate:Case Pairs


```
(∃a-id:A-id) (∃e-id-1:E-id) (∃e-id-2:E-id)
(  a-id ∈ state.A
  ∧ e-id-1 ∈ state.E
  ∧ e-id-2 ∈ state.E
  ∧ a-id.type = operation
  ∧ e-id-1.type = employee
  ∧ e-id-2.type = machine
  ∧ state.R(a-id,agent) = e-id-1
  ∧ state.R(a-id,object) = e-id-2
  ∧ state.C(e-id-1,name) = (names,T.Manhart)
  ∧ state.C(e-id-2,number) = (serial-numbers,NZ745) )
^ (∃a-id:A-id) (∃e-id-1:E-id) (∃e-id-2:E-id)
(  a-id ∈ state.A
  ∧ e-id-1 ∈ state.E
  ∧ e-id-2 ∈ state.E
  ∧ a-id.type = supervision
  ∧ e-id-1.type = employee
  ∧ e-id-2.type = employee
  ∧ state.R(a-id,agent) = e-id-1
  ∧ state.R(a-id,object) = e-id-2
  ∧ state.C(e-id-1,name) = (names,G.Wayshum)
  ∧ state.C(e-id-2,name) = (names,T.Manhart) )
```

Obviously, such logical statements become extremely long for larger relations. One attractiveness of the tabular form of such relations is the compactness of expression of such logical statements.

One feature of the preceding logical statement is that it does not claim that the employee named "T.Manhart" in the supervision association is the same employee as the one named "T.Manhart" in the operation association. As in the previous case involving the problem of uniqueness, the existence of constraints on the relations would allow us to rewrite the previous logical statement to show that the two employees are, in fact, identical.

Another unique feature of the relational data model presented here is the use of the null value ("---") in the relations. An example of the logical

interpretation of a relation row using this value is the following logical statement corresponding to the last row in the relation of Figure 6.5

```
(∃! e-id:E-id)
(  e-id ∈ state.E
  ∧ e-id.type = employee
  ∧ state.C(e-id,name) = (names,G.Wayshum)
  ∧ state.C(e-id,age) = (years,50)
  ∧ ¬ (∃a-id:A-id)
    ( a-id ∈ state.A
      ∧ a-id.type = operate
      ∧ state.R(a-id,agent) = e-id ) ).
```

be employee:object operate:agent		be machine:object operate:object	
employee		machine	
name	age	number	type
names	years	serial-numbers	machine-types
T.Manhart	32	NZ745	lathe
C.Gershag	40	JCL181	press
G.Wayshum	50	---	---

Figure 6.5 - Semantic Relation with Null Value

This states the existence of the employee named "G.Wayshum" and the fact that there is no association with him as operator.

The semantic relation data model will now be described in terms of the

framework presented in Chapter 4.

6.2 The Semantic Relation Application Model Schema

The relational schema describes the headings of the relations as they have been shown in the figures and also states the constraints on the statements allowed in the relations. In set theoretic terms, a semantic relation schema is

$(\text{Rel}, \text{Con}, \text{R-names}, \text{P}, \text{S}, \text{E}, \text{D}, \text{C}, \text{sd}, \text{Str})$

where the components are defined as follows:

Rel: This is a function

$\text{Rel}: \text{R-names} \rightarrow \text{Heading}$

which maps relation names into a heading. A heading is a function of the type

$\text{Heading}: ((\text{PowerSet}'(\text{P} \times \text{S}) \times \text{E}) \cup \text{P}) \rightarrow [\text{C} \rightarrow \text{D}]$

where $\text{PowerSet}'$ is the set of all subsets except the empty set. E is the set of entity set names as used in the semantic graph model. S is the set of possible *semantic case names* which we will assume identical to the set R of role names in the semantic graph model. C is the set of characteristic names and D is the set of domain names as in the semantic graph model. P is the set of *predicates* which contains a predicate for each association type in the graph model plus a "be" predicate for each element of E .

The heading function specifies the "levels" of information shown in the relational headings. The top level is a set of predicate:case pairs or else a single predicate. Then the entity type is specified for the predicate:case pairs. Next the characteristics being given are stated. Each characteristic corresponds to a

single column in the relation's rows. Finally the domain of the characteristic is specified. It is also required that no predicate:case pair may appear in two different predicate:case sets which are in the domain of a relation's heading function. That is, a predicate:case pair may appear only once in the heading.

Con: This is the set of constraints. Each constraint maps the database state into a Boolean value stating whether the state satisfies the constraint. *Further description of constraints will be deferred to Chapter 9.* This is done so that the constraints can be introduced in a coherent manner based on the work presented in Chapter 8.

sd: This function gives domain specifications and is identical to the **sd** used in the semantic graph schema with the exception that in the relational model the null value, "---", is defined to be a valid element of every domain.

R-names, P, S, E, D, C and Str have been previously defined.

AD-A066 386

MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/6 5/2
EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS FOR DATABASE SYS--ETC(U)
JAN 79 S A BORKIN
MIT/LCS/TR-206

UNCLASSIFIED

N00014-75-C-0661

NL

2 OF 4
ADA
066386



6.3 The Semantic Relation Database State

One of the prime advantages claimed for existing "relational-type" data models is the conceptual simplicity of a user seeing the database state in tabular form. This same feature can be seen in the relative simplicity of the definition of the semantic relation database state.

The state is a function from each relation name into a set of tuples (rows) which correspond to the statements expressed by the relation. Formally, the state is a function

$$R\text{-names} \rightarrow \text{Powerset}(\text{Tuples})$$

where Tuples is the set of all possible tuples and is defined as

$$[(\text{Powerset}'(P \times S) \times E) \cup P] \rightarrow [C \rightarrow \text{Domain-element}]$$

where Domain-element is defined as in the graph model except for the addition of the null value.

The above definition simply says that in each row there is a domain element in each column. The Insert-tuples operation (to be defined later) must guarantee that this domain element is in the proper domain for the column as specified by the schema. Note that, formally, the relation heading is present in each tuple.

One other restriction which must be enforced by the operations is that there is no row consisting completely of null values. In the case where there are no associations or entities in the application state which is described by the relation, we will instead let the relation's state be the empty set.

Figure 6.6 shows one possible set of relational headings and the relational state corresponding to the semantic schema and state shown in Figures 5.1 and 5.2. The constraints for this example will be presented in Chapter 9. Remember that in this example both a number and a type are necessary to identify a machine while both machine and shift are necessary to identify an operation association.

Machine-age

be machine:object		
machine		
number	type	age
serial-numbers	machine-types	years
NZ745	lathe	5
JCL181	press	15

Locations-used

operate:object	operate
machine	
location	shift
locations	shifts
1-A	2
1-A	1
2-B	1

Operations

operate:agent	be machine:object operate:object			operate	
employee	machine				
name	number	type	location	shift	cost-per-hour
names	serial-numbers	machine-types	locations	shifts	dollars
T.Manhart	NZ745	lathe	1-A	1	40
B.Belgin	NZ745	lathe	1-A	2	50
C.Gershag	JCL181	press	2-B	1	75

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
---	B.Belgin	35	NZ745	lathe
G.Wayshum	T.Manhart	32	NZ745	lathe
G.Wayshum	C.Gershag	40	JCL181	press
---	G.Wayshum	50	---	---

Figure 6.6 - Semantic Relations

6.4 The Semantic Relation Operation Types

There are only two operation types in the semantic relation data model: Insert-tuples and Delete-tuples. These operation types either add sets of statements or delete sets of statements from relations. A single operation may insert or delete any finite number of tuples into any number of relations in the database state. Even though we speak of "inserting" and "deleting" tuples, it is important to remember that the operations are defined from a purely functional view.

The notation used to describe the relational operations is the same as that used in the definition of the semantic graph operations. Formal specifications are given in Appendix 2. The relevant type domains are defined in Figure A2.1 of Appendix 2. The predefined functions and special notation are given in Figure A2.2. The operations are defined in Figures A2.3 - A2.8. It can be seen that the operations are quite a bit less complex than the semantic graph operations. The ordering definitions in Figure A2.3 will be explained later in this chapter.

6.4.1 Insert-tuples

The predicate function valid-rels defined in Figure A2.4 makes sure that the proposed tuples to be added or deleted are of a proper form as specified by the schema. This includes checking that no tuple consists of all null values.

The operation type Insert-tuples defined in Figure A2.5 maps a schema, a

state and the tuples to be added into a new relational state or the error value. An error results if the tuples to be added are not of the proper form, if any of the tuples to be added already exist in the state, or if any of the constraints evaluate to false when applied to the relations consisting of the unions of the previous state's relational tuples and the tuples to be added. Rather than simple set union, the function s-union ("statement union") defined in Figure A2.6 is used. The way in which s-union differs from ordinary set union will be explained shortly.

Figure 6.7 shows the relations of Figure 2.5. The relational state shown in Figure 6.8 results from

Insert-tuples(schema6.7, state6.7,

(6-1)

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	AB982	lathe

or in set notation

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 6.7 - Complete Semantic Relation State

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press
G.Wayshum	AB982	lathe

Figure 6.8 - Semantic Relations resulting from Operation (6-1)

Insert-tuples(schema6.7, state6.7,

{(Operate,{(((operate,agent)),employee),{(name,G.Wayshum))}},

(6-1)

((((be machine,object),(operate,object)),machine),

{(number,AB982),(type,lathe)})))))

where schema6.7 is the relational schema and state6.7 is the relational state of Figure 6.7. Figure 6.7 does not fully define the schema since the constraints are not presented there. We will present the constraints later and assume that none of the operations shown here violate those constraints.

This operation inserts a single tuple into the Operate relation. Since the resulting state is "state equivalent" (to be defined later) to the graph state of Figure 5.7, operation 6-1 is "operation equivalent" (also to be defined later) to operation 5-3. Note that while we use a colon in the text and figures to separate predicates and cases in predicate:case pairs, when specifying operations we will denote predicate:case pairs as ordered pairs, e.g (operate,agent). Also note that the formal specification in Appendix 2 for Insert-tuples (and Delete-tuples) requires the "insertion" (deletion) of the empty set of statements into each relation for which no change is desired. We will not explicitly show this in either of our operation notations.

The relational state shown in Figure 6.9 is the result of

Insert-tuples(schema6.7, state6.7,

(6-2)

Employees

be employee:object	
employee	
name	age
names	years
B.Belgim	35

or in set notation

Insert-tuples(schema6.7, state6.7,

(6-2)

{(Employees,{(((be employee,object)),employee),
{(name,B.Belgim),(age,35))}})}).

This corresponds to operation 5-1 and the resulting graph state shown in Figure 5.5.

There are several observations which should be made concerning these examples showing operations which insert tuples. Firstly, even though one operation above corresponds to the insertion of a semantic unit in the graph model (6-1) and the other one corresponds to the insertion of an independent entity (6-2), the same operation type is used in the relational model. Secondly, even though a single tuple corresponds to a semantic unit in the Operate relation and to an independent entity in the Employees relation of Figures 6.8 and 6.9, it may be the case that inserting a semantic unit, an independent

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50
B.Belgim	35

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 6.9 - Semantic Relations resulting from Operation (6-2)

entity or an independent association requires the addition of several tuples.

Finally, it should be noted that a single Insert-tuples operation could do the equivalent work of several semantic graph operations. For example, Figure 6.10 shows the result of

Insert-tuples(schema6.7, state6.7,

(6-3)

Employees

be employee:object	
employee	
name	age
names	years
B.Belgin	35

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	AB982	lathe

or in set notation

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50
B.Belgim	35

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press
G.Wayshum	AB982	lathe

Figure 6.10 - Semantic Relations resulting from Operation (6-3)

Insert-tuples(schema6.7, state6.7,

{(Employees,((((be employee,object)),employee),

{(name,B.Belgim),(age,35))})),

(6-3)

(Operate,((((operate,agent)),employee),{(name,G.Wayshum)}),

((((be machine,object),(operate,object)),machine),

{(number,AB982),(type,lathe))})).

This is equivalent to the composition of the Insert-semantic-unit and Insert-ind-entity graph operations, 5-1 and 5-3 which created Figures 5.5 and 5.7.

The following example will show how the s-union function differs from ordinary set union. This difference is only visible when tuples with null values ("---") are in the relations. If this is the case, then it might be desired to add to the relation a statement which is identical to some existing statement in the relation with the exception that some of the values which were null in the existing statement are non-null in the added statement. When this is the case, the existing tuple is replaced by the added tuple. That is, the existing tuple is automatically deleted.

Semantically, this means that a statement of the form "there is no entity(association) such that . . ." is automatically deleted when the statement is made false. For example, Figure 6.11 is the result of

Insert-tuples(schema6.6, state6.6,

(6-4)

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	NZ745	lathe

or in set notation

Insert-tuples(schema6.6, state6.6,

{(Supervise-operate,(((supervise,agent)),employee),{(name,G.Wayshum)}),

((((be employee,object),(supervise,object),(operate,agent)),

(6-4)

employee),{(name,B.Belgim),(age,35)}),

((((operate,object)),machine),{(number,NZ745),(type,lathe)})))).

This operation is equivalent to the addition of the independent association in operation 5-2 which resulted in Figure 5.6.

This type of operation might be better understood, and is formally defined, in terms of a *partial ordering* of statements. This ordering is used in the s-union and s-difference functions. The ordering says that a statement asserting the existence of an entity or association *takes precedence over* ("is greater than", >) a statement asserting that no such entity or association exists. So

Machine-age

be machine:object		
machine		
number	type	age
serial-numbers	machine-types	years
NZ745	lathe	5
JCL181	press	15

Locations-used

operate:object	operate
machine	
location	shift
locations	shifts
1-A	2
1-A	1
2-B	1

Operations

operate:agent	be machine:object operate:object			operate		
	machine					
employee	name	number	type	location	shift	cost-per-hour
names	serial-numbers	machine-types	locations	shifts	dollars	
T.Manhart	NZ745	lathe	1-A	1	40	
B.Belgim	NZ745	lathe	1-A	2	50	
C.Ger'shag	JCL181	press	2-B	1	75	

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	NZ745	lathe
G.Wayshum	T.Manhart	32	NZ745	lathe
G.Wayshum	C.Gers'ha	40	JCL181	press
---	G.Wayshum	50	---	---

Figure 6.11 - Semantic Relations resulting from Operation (6-4)

when a statement asserting existence is added to a relation, the statement asserting the non-existence is automatically deleted.

The ordering of statements is formally defined in Figure A2.3. This ordering is based on the definition that within a given domain, all non-null values are greater than the null value. Furthermore, non-null values are incomparable to any value other than the null value and itself (to which it is equal).

Extending this to tuples with equal headings, we define, for example, that the tuple

<i>supervise:agent</i>	<i>supervise:object</i> <i>operate:agent</i>	<i>operate:object</i>
employee	employee	machine
<i>name</i>	<i>name</i>	<i>number</i>
names	names	serial-numbers
G.Wayshum	B.Belgin	NZ745

(6-5)

is greater than

supervise:agent	supervise:object operate:agent	operate:object
employee	employee	machine
<i>name</i>	<i>name</i>	<i>number</i>
names	names	serial-numbers
---	B.Belgin	NZ745

(6-6)

Tuple 6-5 is incomparable with

supervise:agent	supervise:object operate:agent	operate:object
employee	employee	machine
<i>name</i>	<i>name</i>	<i>number</i>
names	names	serial-numbers
G.Wayshum	C.Gershag	JCL181

(6-7)

By this definition, the first row of the relation Supervise-operate in Figure 6.11, inserted by operation 6-4, takes precedence over the first row of Supervise-operate in Figure 6.6.

It should also be noted that whenever a single characteristic in a tuple is null, every other characteristic describing the same entity or predicate must be null. This is in accordance with the interpretation of the null value. The constraints associated with the database schema must assure this property. The function valid-rels also checks that no two tuples in a single relation are

comparable under this ordering.

6.4.2 Delete-tuples

For use with the Delete-tuples operation, the ordering of statements allows the comparison of tuples in which the predicates in one tuple are a subset of the predicates in another tuple. This involves the ordering definition on tuple headings in Figure A2.3. In particular,

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	AB982	lathe

(6-8)

is greater than

operate:agent	operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	AB982	lathe

(6-9)

Note that

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
---	AB982	lathe

(6-10)

is incomparable to tuple 6-9. While the heading in 6-9 is less than the heading in 6-10, the corresponding values in 6-9 are not all less than or equal to those in 6-10 (since "G.Wayshum" is greater than "---").

The operation type Delete-tuples defined in Figure A2.7 maps a schema, a state and the tuples to be deleted into a new relational state or the error value. Each tuple specified in the deletion must be less than or equal to some tuple in the database state. An error results if any of the constraints evaluate to false

when applied to the relations consisting of the difference between the previous state's relational tuples and the tuples to be deleted. For reasons similar to those resulting in the use of s-union instead of set union, the function s-difference defined in Figure A2.8 is used by Delete-tuples instead of regular set difference. The first two examples specify the deletion of tuples equal to tuples in the relations of concern. In this case, s-difference is identical to regular set difference.

The operation

Delete-tuples(schema6.8, state6.8,

(6-11)

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	A8982	lathe

or in set notation

Delete-tuples(schema6.8, state6.8,

{(Operate,{(((operate,agent)),employee),{(name,G.Wayshum))),

(6-11)

{(((be machine,object),(operate,object)),machine),

{(number,AB982),(type,lathe))}}))

results in the relational state shown in Figure 6.7. This is equivalent to the deletion of a semantic unit in operation 5-6.

Figure 6.7 is also the result of

Delete-tuples(schema6.9, state6.9,

(6-12)

Employees

be employee:object	
employee	
name	age
names	years
B.Belgim	35

or in set notation

Delete-tuples(schema6.9, state6.9,

(6-12)

{(Employees,{(((be employee,object)),employee),

{(name,B.Belgim),(age,35))}}))

The next example presents a situation in which the heading in a tuple

specified for deletion causes the tuple to be "less than" a tuple in the database state. For the operation to be legal, we must assume that the constraints associated with Figure 6.8 do not require that every machine have an operator. "operation" can then be considered to be an independent association rather than part of a semantic unit.

Figure 6.12 is the result of

Delete-tuples(schema6.8, state6.8,

(6-13)

Operate

operate:agent	operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
G.Wayshum	AB982	lathe

or in set notation

Delete-tuples(schema6.8, state6.8,

{(Operate,((((operate,agent)),employee),((name,G.Wayshum))),
(((operate,object)),machine),
{(number,AB982),(type,lathe))}})).

(6-13)

In this operation the tuple specified for deletion is less than the third

Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press
---	AB982	lathe

Figure 6.12 - Semantic Relations resulting from Operation (6-13)

tuple of Operate in Figure 6.8. While the values in the tuples are equal, the predicates in the heading of the tuple in the operation are a subset of the predicates in the relation.

The meaning of operation 6-13 is that the "operation" association between G.Wayshum and machine AB982 is to be deleted, but the existence of machine AB982 is to remain a part of the database state. This should be contrasted with operation 6-11 in which the machine and association were deleted together.

The mechanism of such deletions is to reduce the precedence of tuples in the database state until they become less than or incomparable to the tuples specified in the deletion. That is, each assertion in the tuples specified for deletion must be made false.

This is done by the introduction of null values. For a tuple in the database state which is greater than a specified deletion, null values are introduced for each characteristic which is equal in value to that in the deletion and which describes precisely the same predicate or set of predicate:case pairs.

Consider this rule applied to operation 6-13 on a column by column basis. The tuple in the deletion is less than the third tuple in the Operate relation of Figure 6.8. The first columns have equal values, "G.Wayshum", and describe the same predicate:case pair, "operate:agent". So for the resulting tuple, this column will have the null value.

The second and third columns have equal values, but do not describe the identical predicate:case pairs. The tuple in the deletion describes "operate:object"

while the tuple in the database state describes both "operate:object" and "be machine:object". So these two values will not be changed. The resulting tuple is the third tuple of Operate in Figure 6.12. As shown in 6-9 and 6-10, this tuple is incomparable to the tuple in the delete operation.

Figure 6.6 is the result of

Delete-tuples(schema6.11, state6.11,

(6-14)

Supervise-operate

supervise:agent	supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	---	---

or in set notation

Delete-tuples(schema6.11, state6.11,

{{(Supervise-operate, {(((supervise, agent)), employee), {(name, G.Wayshum)}},

(6-14)

{(((supervise, object), (operate, agent)), employee),

{(name, B.Belgim), (age, 35)}},

{(((operate, object)), machine), {(number, "---"), (type, "---)}}})).

This operation, which is equivalent to the deletion of an independent association

in operation 5-5, makes use of null values. As in the previous example, the result of the s-difference function is not the same as set difference because the tuple in operation 6-13 is "less than" the first row of Supervise-operate in Figure 6.11.

6.5 Definitions of Valid Operations and Valid States

As in the case of the semantic graph data model, we want to define the set of operations for a given schema which map database states into database states.

Definition 6.1: For a given $r\text{-schema}_n \in R\text{-schema}$, the set of valid operations is the union of the following sets for $1 \leq i \leq 2$:

$\{\text{operation}_{ij}(r\text{-state}:R\text{-state})$

returns $R\text{-state}$

$\text{return Op-type}_i(r\text{-schema}_n, r\text{-state}, \text{arg}_j)\}$

where $\text{arg}_j \in R\text{-state}$

and for:

$i = 1$: Op-type_i = Insert-tuples

$i = 2$: Op-type_i = Delete-tuples

The set of valid semantic relation database states correspond to those elements of the set $R\text{-state}$ which represent possible states of the application. As in the case of the semantic graph data model, we will require that the

initial state for all semantic relation applications be the empty state.

Definition 6.2: The set of valid semantic relation states for a given semantic relation schema r -schema and set of operations R -operations, where R -operations is defined for r -schema as in Definition 6.1, is defined as follows:

1. The empty state r -state which maps each relation name defined in r -schema into the empty set is a valid semantic relation state if for all constraints $con \in r\text{-schema.Con}$, $con(r\text{-state}) = \text{true}$.

2. If r -state is a valid state, $r\text{-op} \in R\text{-operations}$ and $r\text{-op}(r\text{-state}) \neq \text{error}$,
then $r\text{-state}' = r\text{-op}(r\text{-state})$ is a valid semantic relation state.

Chapter 7 - Data Model Equivalence: Definitions

Now that the semantic graph and semantic relation data models have been defined, we will begin the examination of their equivalence properties. Much as Chapter 4 presented a general framework for data model definition, this chapter gives definitions of equivalence properties applicable to any data models defined in terms of the Chapter 4 framework. Examples, however, are drawn from the semantic graph and semantic relation data models. An informal definition of database state equivalence for the semantic graph and semantic relation data models is also presented.

The notion of *data model equivalence* is based on *application model equivalence* which is in turn based on *database state equivalence*. We will examine these in a "bottom-up" manner.

7.1 Database State Equivalence

We would like to say that two database states expressed in terms of two different data models are equivalent if they represent identical application states. However, the precise specification of state equivalence depends on the nature of the two data models. That is, what is the *interpretation* of the database state in terms of the application?

The ease with which such interpretations can be given varies widely between data models. As we saw in Chapter 3, attempts to define equivalence

properties for syntactic data models have been based on rather arbitrary mapping definitions. The use of semantic data models can ease the task of defining equivalent database states.

We have claimed that many of the semantic graph and semantic relation database states shown in Chapters 5 and 6 are equivalent. We could show this by translating each relational statement into a formal logic statement and then showing that the semantic graph state is a *model*, in the formal logic sense, for the set of logical statements. A prerequisite to this would be "agreement" between the semantics of the two data models. Chapters 8, 9 and 10 will consider these problems in detail for the semantic graph and semantic relation data models.

For example, a mapping between association types and the predicate used to express information concerning each association type would be required ("supervision" and "supervise", "operation" and "operate"). That is, there must be a translation between the natural language case grammars on which the two data models are based. Hopefully, *the natural language meaning of the words would help confirm that the mappings are done correctly.* At this point we give a definition of state equivalence for semantic graph and semantic relation database states.

Definition 7.1: Assuming that each semantic graph or semantic relation database describes only identifiable entities and associations, then two database states, each of which is based on either the semantic graph or semantic relation data model, are equivalent if and only if

1. For both states, only one value is specified for a single characteristic of an entity or association.
2. For both states, only one entity may be specified as filling a given role in an association.
3. The states define equal sets of entity and association identifiers.
4. For each entity, the same characteristics are defined in each state and are of equal value.
5. For each association, the same characteristics are defined in each state and are of equal value.
6. For each association, the same roles are defined in each state and are filled by the same entity.

Qualifications 1 and 2 of Definition 7.1 are aimed at semantic relation database states. Since information may be stated redundantly (e.g. in several relations), this qualification guarantees that there will be no inconsistencies within the database state.

As a basis for the examples of this chapter, consider the semantic graph schema of Figure 7.1. This is the schema of Figure 2.1 with the addition of the "supervision" association. Figure 7.2 is a semantic graph state corresponding to this schema. Figure 7.3 shows an "equivalent" (assuming proper constraints) semantic relation schema and state.

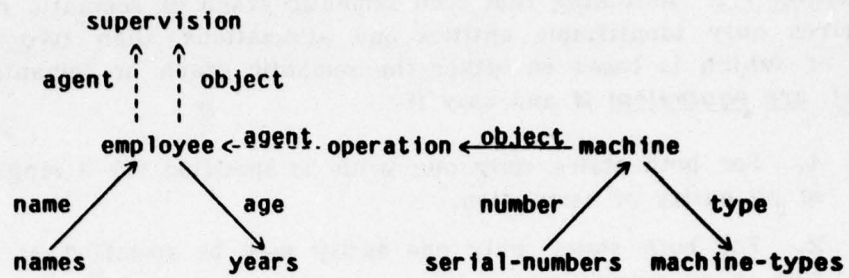


Figure 7.1 - Semantic Graph Schema

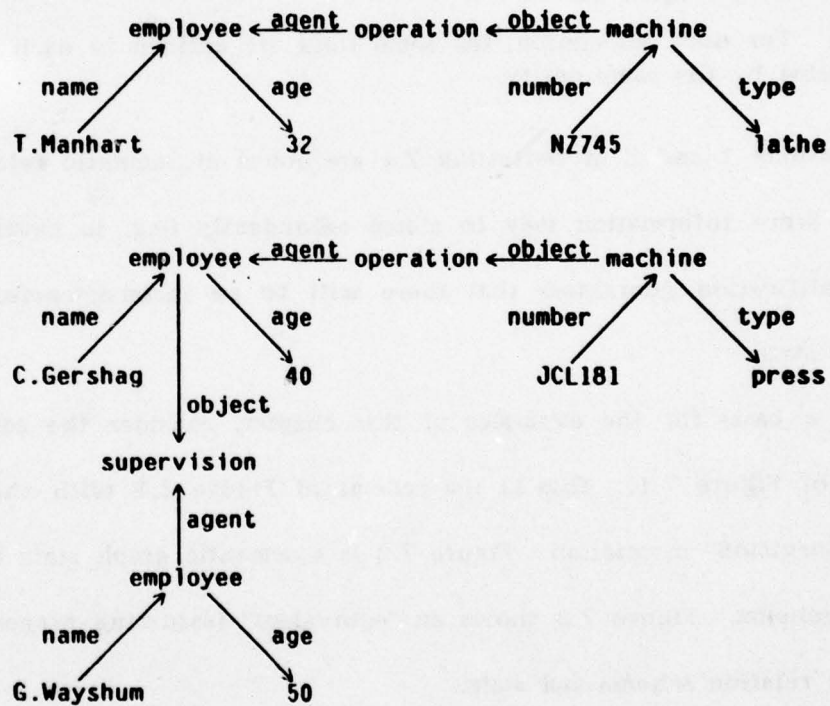


Figure 7.2 - Semantic Graph Database State

be employee:object		Employees
employee		
name	age	
names	years	
T.Manhart	32	
C.Gershag	40	
G.Wayshum	50	

operate:agent	be machine:object operate:object		Operate
employee	machine		
name	number	type	
names	serial-numbers	machine-types	
T.Manhart	NZ745	lathe	
C.Gershag	JCL181	press	

supervise:agent	supervise:object operate:agent	operate:object	Jobs
employee	employee	machine	
name	name	number	
names	names	serial-numbers	
G.Wayshum	C.Gershag	JCL181	
----	T.Manhart	NZ745	

Figure 7.3 - Semantic Relations

7.2 Operation and Application Model Equivalence

Now we will return to the general problem of defining data model equivalence for any two data models. Recall that a data model is a set of application models. We will consider two data models

$$M = \{m\text{-model}_1, \dots, m\text{-model}_m\}$$

and

$$N = \{n\text{-model}_1, \dots, n\text{-model}_n\}.$$

Our goal is then to specify under what conditions we can say that M is equivalent to N .

It is necessary to define *application model equivalence*. That is, how to identify when application models from different data models represent the same application. We will consider defining application model equivalence for two application models $m\text{-model} \in M$ and $n\text{-model} \in N$. We will assume that $M\text{-ops}$ and $N\text{-ops}$, and $M\text{-states}$ and $N\text{-states}$ define the sets of allowable operations and the sets of valid states for these two application models. We will refer to operations and database states from these sets as $m\text{-op}$ and $n\text{-op}$, and $m\text{-state}$ and $n\text{-state}$, respectively.

We will assume that a state equivalence correspondence has been defined between $M\text{-states}$ and $N\text{-states}$. That is, given states $m\text{-state}$ and $n\text{-state}$, we can tell whether or not they represent the same application state. We require that the state equivalence correspondence be *onto* both $M\text{-states}$ and $N\text{-states}$. That is, the "expressive power" of the state representation schemes are equal in

both application models. It also seems reasonable to assume that for both application models, some specific application state is represented by a unique state. Hence, we require state equivalence to be a 1-1 onto correspondence.

This does not require that there be a 1-1 mapping between components of equivalent states as required, for example, by the mapping of each relational tuple to either a DBTG record or a set membership link in [Zimmerman]. The specific manner of defining the state equivalence relation is dependent upon the data model being considered.

A straight forward approach to defining application model equivalence is to require a 1-1 onto correspondence, *operation equivalence*, between the sets of operations $M\text{-ops}$ and $N\text{-ops}$ such that state equivalence and operation equivalence form an isomorphism. We make the assumption that the *error* states of all application models are equivalent.

First we define *operation equivalence*.

Definition 7.2: A function

$$m : M\text{-states} \rightarrow M\text{-states}$$

is operation equivalent to a function

$$n : N\text{-states} \rightarrow N\text{-states}$$

if and only if for any two equivalent states $m\text{-state} \in M\text{-states}$ and $n\text{-state} \in N\text{-states}$, $m(m\text{-state})$ is state equivalent to $n(n\text{-state})$.

Then we define *isomorphic application model equivalence*.

Definition 7.3: Two application models *m-model* and *n-model* are isomorphically equivalent if and only if:

i). For every operation $m\text{-op} \in M\text{-ops}$, there exists exactly one $n\text{-op} \in N\text{-ops}$ which is operation equivalent to $m\text{-op}$.

and ii). For every operation $n\text{-op} \in N\text{-ops}$, there exists exactly one $m\text{-op} \in M\text{-ops}$ which is operation equivalent to $n\text{-op}$.

This is the "most strict" form of application model equivalence we will define. However, it may in fact be too strict. Some application models which we would like to call equivalent do not satisfy the definition. In particular, consider the semantic relation and semantic graph data models. Suppose that in the semantic relation data model we consider an operation which inserts tuples corresponding to the insertion in the graph model of several independent entities. That is, the relational operation is equivalent to the composition of several graph operations.

To account for this form of equivalence, we allow the definition of operation equivalence to be applied to the sets $M\text{-op}^*$ and $N\text{-op}^*$ which represent the set of all compositions of operations in the two application models. This allows the definition of composed operation application model equivalence.

Definition 7.4: Two application models *m-model* and *n-model* are composed operation equivalent if and only if:

i). For every operation $m\text{-op} \in M\text{-ops}$, there exists an operation $n \in N\text{-ops}^*$ which is operation equivalent to $m\text{-op}$.

and ii). For every operation $n\text{-op} \in N\text{-ops}$, there exists an operation $m \in M\text{-ops}^*$ which is operation equivalent to $n\text{-op}$.

Definition 7.4 does not restrict the operation equivalence correspondence to

be 1-1. It does require that there be an operation equivalent to each of the simple (i.e. non-composed) operations of *M-ops* and *N-ops*. By composition of simple operations, the correspondence is onto *M-ops*^{*} and *N-ops*^{*}.

In practical terms, we would hope that the operation equivalence mappings can be expressed as an algorithm rather than an explicit enumeration of an extremely large number of equivalent pairs. It is such an algorithm which would actually allow the implementation of a database system which provides users of two different data models with access to the "same" data.

Note that based on the types of operation equivalence introduced so far, the translation of operations from one application model to an equivalent application model can be done independently of the database state. That is, operations are equivalent for all possible states. Consequently, such a translation could be done at "compile-time".

This is not necessarily always the case. The mappings of equivalent operations may be dependent upon the database state. Consider the case of adding to the graph database state of Figure 7.2 a supervision association between G.Wayshum and T.Manhart resulting in Figure 7.4. The equivalent semantic relation database state has the same Employees and Operate relations shown in Figure 7.3 with the new Jobs relation shown in Figure 7.5.

The semantic relation operation equivalent to the stated graph operation consists of the insertion of the second tuple in the Jobs relation of Figure 7.5. There are two important things to notice about this operation.

Firstly, we did not explicitly delete the second tuple of the Figure 7.3

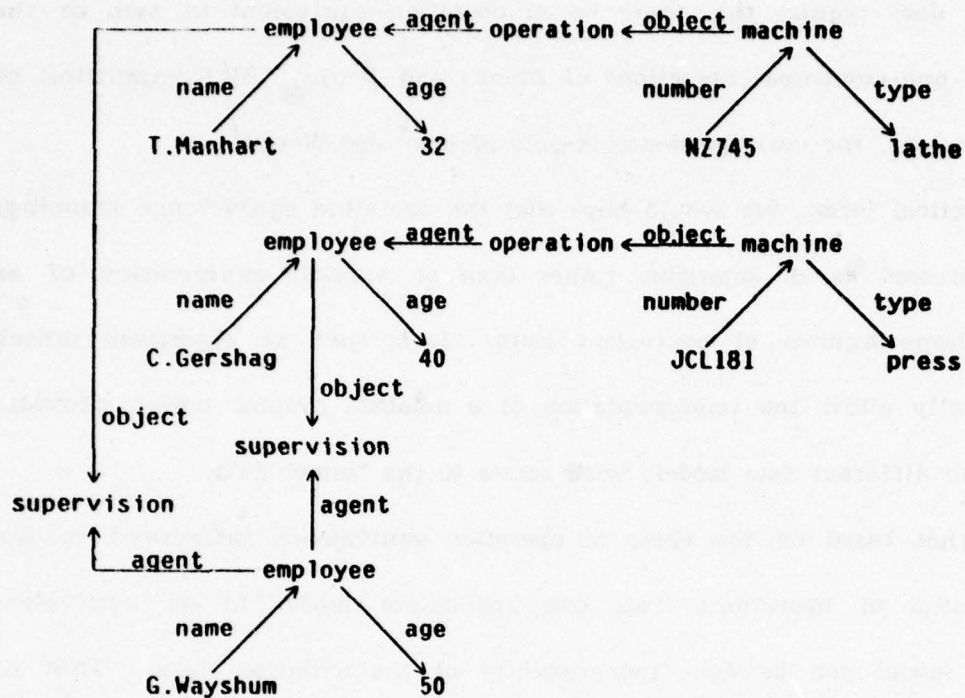


Figure 7.4 - Semantic Graph Database State after Insertion

Jobs relation. This is because the semantic relation Insert-tuples operation type is defined (in Chapter 6) to automatically delete all tuples in a relation "less than" those inserted. The partial ordering of tuples is based on all non-null domain values being greater than null ("----") and incomparable with any values other than null and itself.

More importantly, the values in the added tuple are dependent upon the database state of Figure 7.3. Suppose that the semantic graph state of Figure 7.4 had no operation association involving T.Manhart. This would not change the graph operation needed to insert the supervision association between

supervise:agent	supervise:object operate:agent	operate:object	Jobs
employee	employee	machine	
<i>name</i>	<i>name</i>	<i>number</i>	
names	names	serial-numbers	
G.Wayshum	C.Gershag	JCL181	
G.Wayshum	T.Manhart	NZ745	

Figure 7.5 - Semantic Relation after Insertion

T.Manhart and G.Wayshum.

In the case of the relational model, the lack of an operation association for T.Manhart would change which tuple needed to be added to reflect the new supervision. The tuple to be added would be the second tuple in Figure 7.6 rather than the second tuple in Figure 7.5. [Klug and Tsichritzis] have also pointed out the possibility of the state affecting the definition of equivalent operations.

We reflect the possibility of state dependence in the following definitions of state dependent operation and application model equivalences:

supervise:agent	supervise:object operate:agent	operate:object	Jobs
employee	employee	machine	
name	name	number	
names	names	serial-numbers	
G.Wayshum	C.Gershag	JCL181	
G.Wayshum	T.Manhart	----	

Figure 7.6 - Semantic Relation showing State Dependence of Insertion

Definition 7.5: A function

$$m : M\text{-states} \rightarrow M\text{-states}$$

and a function

$$n : N\text{-states} \rightarrow N\text{-states}$$

are state dependent operation equivalent for a given pair of equivalent states $m\text{-state} \in M\text{-states}$ and $n\text{-state} \in N\text{-states}$, if and only if $m(m\text{-state})$ is state equivalent to $n(n\text{-state})$.

Definition 7.6: Two application models $m\text{-model}$ and $n\text{-model}$ are state dependent equivalent if and only if for every pair of equivalent states $m\text{-state} \in M\text{-states}$ and $n\text{-state} \in N\text{-states}$,

- i). For every operation $m\text{-op} \in M\text{-ops}$, there exists an operation $n \in N\text{-ops}$ which is state dependent operation equivalent to $m\text{-op}$ for $m\text{-state}$ and $n\text{-state}$.
- and ii). For every operation $n\text{-op} \in N\text{-ops}$, there exists an operation $m \in M\text{-ops}$ which is state dependent operation equivalent to $n\text{-op}$ for $m\text{-state}$ and $n\text{-state}$.

The types of application model equivalence defined are decreasingly strict. That is, isomorphic equivalence implies composed operation equivalence, and composed operation equivalence implies state dependent equivalence. There are

other possibilities - we could define state dependent equivalence which does not allow composed operations. The intuition guiding the definitions presented here is that as data models become more dissimilar, they do so in several ways simultaneously. This seems justified by the examples considered.

7.3 Data Model Equivalence

Finally we reach the goal of defining data model equivalence.

Definition 7.7: Two data models M and N are (isomorphically, composed operation, state dependent) equivalent if and only if (isomorphic, composed operation, state dependent) application model equivalence defines a correspondence between M and N which is onto both M and N .

That is, if two data models are equivalent, then for any application model in one data model there is an equivalent application model in the other data model. The expressive power of the data models are equivalent. The definition is dependent upon the type of application model equivalence required.

Partial equivalence results if when considering two data models, there are some equivalent application models, but not all application models in one or both data models have equivalent application models in the other data model. As will be seen in the next chapter, the semantic relation and the semantic graph data models are only partially equivalent.

Note that the definitions do not require that application model equivalence be 1-1. In the case of the semantic relation and graph data models, there may be several relational application models state dependent equivalent to each graph

model. This corresponds to the many different ways of grouping into relations the statements corresponding to a single graph state. For example, Figure 7.9 (with appropriate constraints) shows a single semantic relation which, like Figure 7.3 (consisting of three relations), is application model equivalent and state equivalent to Figures 7.1 and 7.2. Such a property makes the semantic relation model a good candidate for the [ANSI] external data model allowing many different relational views of a single semantic graph conceptual application model.

Machine-shop

supervise:agent	be employee:object supervise:object operate:agent		be machine:object operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	C.Gershag	40	JCL181	press
---	T.Manhart	32	NZ745	lathe
---	G.Wayshum	50	---	---

Figure 7.9 - Equivalent Semantic Relation

Chapter 8 - The Restricted Semantic Relation Data Model

From the discussion and examples of Chapter 7, it is clear that the semantic graph and semantic relation data models are not isomorphically or composed operation equivalent. We will state here that they are *also not state dependent equivalent*. While every semantic graph application model has a state dependent equivalent semantic relation application model, not every semantic relation application model has an equivalent semantic graph application model. The basic reason for the existence of such application models has to do with the great freedom allowed in defining constraints. While we will not develop the formal notation for specifying constraints until Chapter 9, we will still describe here an example of a semantic relation application model with no semantic graph equivalent.

Consider the semantic graph application model in Figure 8.1 (identical to Figure 5.1) and the semantic relation application model (less constraints) of Figure 8.2 (identical to Figure 6.6). We have previously stated that with "appropriate" constraints these two application models are equivalent. Let us consider an "inappropriate" constraint.

In particular, assume the specification of a functional constraint in the Supervise-operate relation of Figure 8.2 between the first column (employee supervisor names) and the last column (machine types). This constraint says that the types of every machine operated by the employees of a given supervisor must be equal. While this might be a reasonable restriction, it

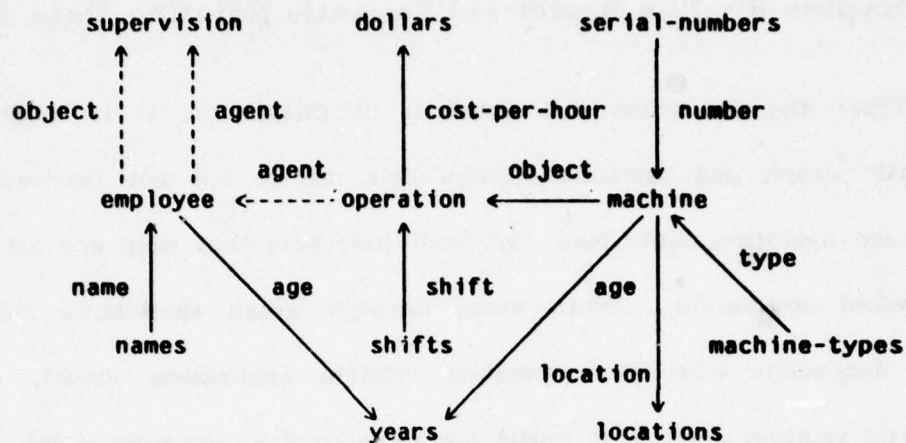


Figure 8.1 - More Complex Semantic Graph Schema

simply cannot be expressed by a semantic graph schema. Note that it is *not* satisfactory to make "machine-type" a characteristic of employee.

Firstly, there would not be an appropriate value for this characteristic for those employees not supervising machine operators. Secondly, there would be no place to record the machine type for machines whose operators have no supervisor. In light of this problem, we would like to identify those semantic relation application models which are state dependent equivalent to some semantic graph model. We will do this by giving algorithms for deriving semantic relation application models from semantic graph application models.

Constraints for the semantic relation data models so derived will be stated in a canonical form. Within the determination of constraint equivalence, this algorithm will define the complete set of semantic relation application models which have semantic graph equivalents. The set of resulting semantic relation

Machine-age

be machine:object		
machine		
number	type	age
serial-numbers	machine-types	years
NZ745	lathe	5
JCL181	press	15

Locations-used

operate:object	operate
machine	
location	shift
locations	shifts
1-A	2
1-A	1
2-B	1

Operations

operate:agent	be machine:object operate:object			operate	
employee	machine				
name	number	type	location	shift	cost-per-hour
names	serial-numbers	machine-types	locations	shifts	dollars
T.Manhart	NZ745	lathe	1-A	1	40
B.Belgim	NZ745	lathe	1-A	2	50
C.Gershag	JCL181	press	2-B	1	75

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
---	B.Belgim	35	NZ745	lathe
G.Wayshum	T.Manhart	32	NZ745	lathe
G.Wayshum	C.Gershag	40	JCL181	press
---	G.Wayshum	50	---	---

Figure 8.2 - Semantic Relations

application models will form the *restricted semantic relation data model*.

This chapter presents an algorithm for specifying the possible sets of relations which may form the basis for semantic relation application models state dependent equivalent to a given semantic graph application model. Note that within the [ANSI] framework this basic procedure could be considered a database design methodology for deriving multiple external semantic relation schemas corresponding to a single semantic graph conceptual schema.

Chapter 9 completes the specification of the restricted semantic relation data model. The bulk of Chapter 9 is an algorithm for specifying the relational constraints necessary for the restricted semantic relation application models. Formal definitions of the constraint types are introduced as necessary in Chapter 9. Chapter 10 proves that the semantic graph and restricted semantic relation data models are state dependent equivalent.

The rest of the present chapter presents Algorithm 8.1. Examples are interspersed with the algorithm definition.

Algorithm 8.1: The following algorithm describes how to derive all but the constraint component Con of a relational schema

$$r\text{-schema} = (\text{Rel}, \text{Con}, \text{R-names}, \text{P}, \text{S}, \text{E}, \text{D}, \text{C}, \text{sd}, \text{Str}),$$

from a given semantic graph schema

$$g\text{-schema} = (\text{E}, \text{A}, \text{D}, \text{r}, \text{c}, \text{fr}, \text{fc}, \text{sd}, \text{R}, \text{C}, \text{Str}).$$

The only restriction on g-schema is that no element of A is of the form be *type* where *type* \in E. This is due to the special meaning given in the semantic

relation data model to this predicate. If such an association exists, we will simply assume it is renamed for purposes of the algorithms in Chapters 8-10.

In outline, the algorithm is as follows:

I. Define an *augmented graph schema*. This adds the special predicates be *type* as nodes in the semantic graph schema g-schema.

II. Select subgraphs of the augmented schema defining that portion of the application which will be described in each relation, making sure that four types of requirements are satisfied.

III. Define the relational schema components.

A. Define P, S, E, D, C, sd and Str.

B. Define the relation headings and names, Rel and R-names, by defining a relation corresponding to each subgraph defined in step II as follows:

1. Partition the subgraph corresponding to the grouping of the columns by predicate:case pairs as desired in the relation's heading.
2. Define the relation's heading for columns describing predicate:case pairs.
3. Define the relation's heading for columns describing predicate characteristics.
4. Combine the results of steps 2. and 3.
5. Associate a relation name with the resulting heading.

For the reader desiring only a high level view, the four requirements specified in step II. which must be satisfied by the subgraphs (i.e. the relations) should be examined in addition to the outline above.

The algorithm itself follows:

I. Define an *augmented graph schema*. This adds the special predicates *be type* as nodes in the semantic graph schema *g-schema*. These nodes are added so that once transformed into relations, the existence of entities can be stated independently of their participation in associations.

Let

$$g\text{-schema}' = (E', A', D', r', c', fr', fc', sd', R', C', Str'),$$

where:

$$1. A' = A \cup \{(be\ type, object) \mid type \in E\}$$

This adds the *be type* predicate for each entity type to the set of associations.

$$2. r' = r \cup \{((be\ type, object), type) \mid type \in E\}$$

This adds a role set edge connecting the new association set node to the proper entity set node.

$$3. fr' = fr \cup \{((be\ type, object), (restricted, mandatory)) \mid type \in E\}$$

This states that every entity of the given type must belong to exactly one "be type" association.

$$4. E' = E$$

$$5. D' = D$$

$$6. c' = C$$

$$7. fc' = fc$$

$$8. sd' = sd$$

$$9. R' = R$$

$$10. C' = C$$

$$11. Str' = Str$$

Figure 8.3 shows a semantic graph schema identical to Figure 2.1. Figure 8.4.a shows such an augmented schema for the semantic graph schema of Figure 8.3 while Figure 8.4.b shows the augmented schema for the semantic graph schema of Figure 8.1.

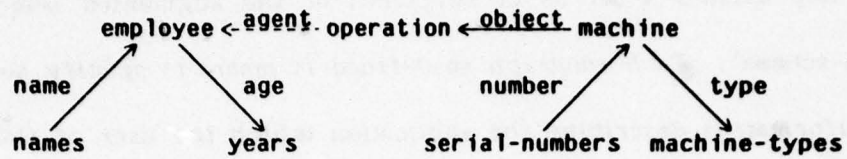
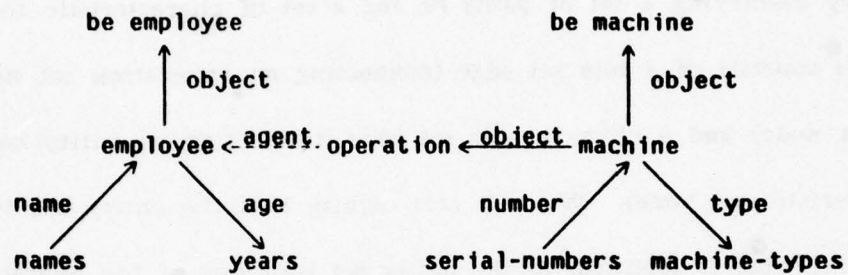
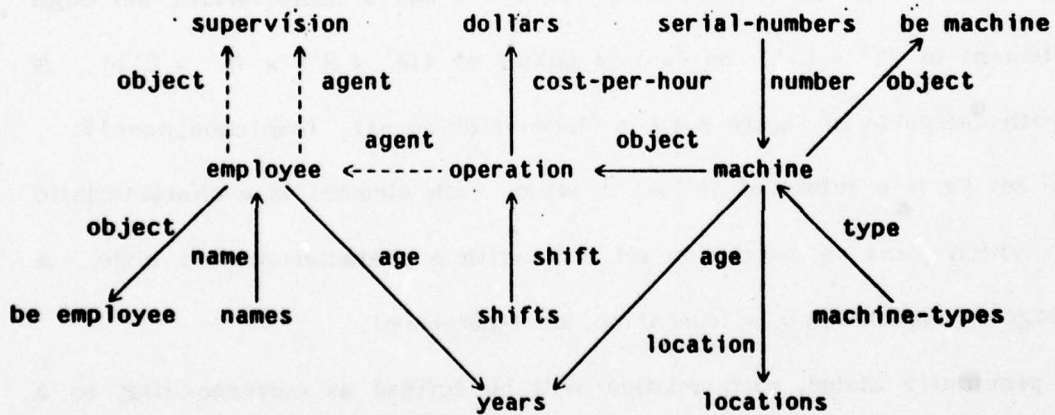


Figure 8.3 - Semantic Graph Schema



a. Augmentation of Figure 8.3



b. Augmentation of Figure 8.1

Figure 8.4 - Augmented Semantic Graph Schemas

II. This step defines a set SG of subgraphs of the augmented semantic graph schema g -schema'. Each subgraph so defined is meant to specify some portion of the information describing the application which the user of the relational database system would like to see displayed in a single relation.

Informally, subgraphs will be delineated by encircling a portion of the augmented semantic graph schema of concern. Formally, each subgraph will be defined by specifying a set of paths Pa and a set of characteristic set edges Ca . Each path consists of a role set edge (connecting an association set node and an entity set node) and a characteristic set edge (connecting an entity set node and a characteristic set node). We will also require that the entity set terminus of the role set edge is identical to the entity set terminus of the characteristic set edge. That is, the two edges form a continuous path in the schema. We specify a role set edge by an element of $(A' \times R')$ and a characteristic set edge by an element of $(E' \times C')$. So Pa is a subset of $((A' \times R') \times (E' \times C'))$. A sample path definable in Figure 8.4.a is $((operation, agent), (employee, name))$.

The set Ca is a subset of $(A' \times C')$ where each element is a characteristic set edge which joins an association set node with a characteristic set node. A sample edge in Figure 8.4.b is $(operation, cost-per-hour)$.

As previously stated, each relation will be defined as corresponding to a particular subgraph. Figure 8.5 shows the semantic relations of Figure 2.5. Figure 8.6 shows the two subgraphs (of the Figure 8.4.a augmented semantic graph schema) corresponding to the two relations of Figure 8.5. Figure 8.6.a shows the "view" of the application shown in the Employees relation of Figure

8.5 while Figure 8.6.b shows the "view" of the application shown in the Operate relation. The paths in Pa and edges in Ca are shown as heavy numbered lines alongside the edges which make up the paths.

Figure 8.7 shows the four subgraphs (of the Figure 8.4.b augmented semantic graph schema) corresponding to the four relations of Figure 8.2. Figures 8.7.a, b, c and d correspond respectively to the relations Locations-used, Machine-age, Operation and Supervise-operate.

To ensure that the set of relations corresponding to the specified set of subgraphs completely describe the application (and hence is equivalent to the semantic graph schema), several requirements must be fulfilled by the choice of subgraphs. One subgraph may be used to fulfill any number of requirements.

These four requirements are:

1. Existence and Identity. For each association set node $a \in A'$ there exists at least one subgraph $sg \in SG$ such that:
 - a. For each role r such that $fr'(a,r).restrict = restricted$, for each characteristic c where $fc'(e,c) = key$ and $e = R'(a,r)$, there is a path $((a,r), (e,c))$ in the set Pa of sg .
 - b. For each characteristic c such that $fc'(a,c) = key$, there is a characteristic edge (a,c) in the set Ca of sg .

This requirement implies that for each association, at least one relation will contain all of the characteristics needed to uniquely identify the association. This requirement also implies that for each entity type, there will be at least one relation with the predicate be *type* which has all of the key characteristics of the entity type.

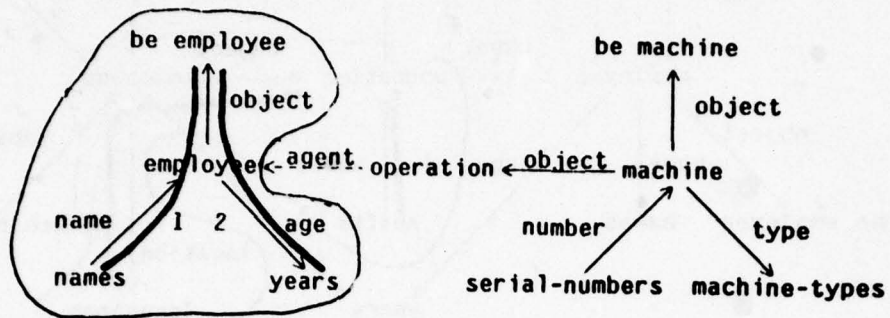
Employees

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

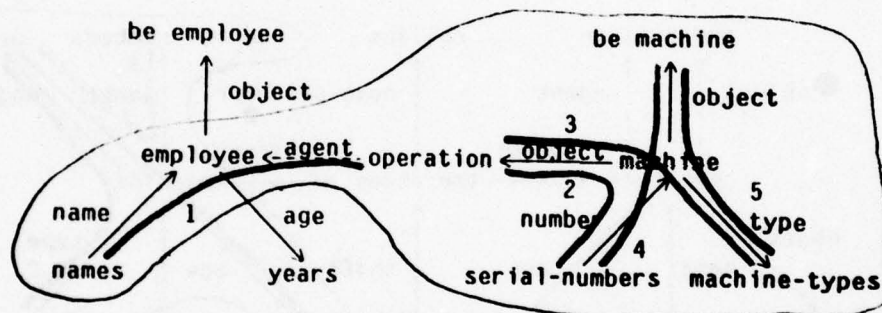
Operate

operate:agent	be machine:object operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
T.Manhart	NZ745	lathe
C.Gershag	JCL181	press

Figure 8.5 - Complete Semantic Relation State

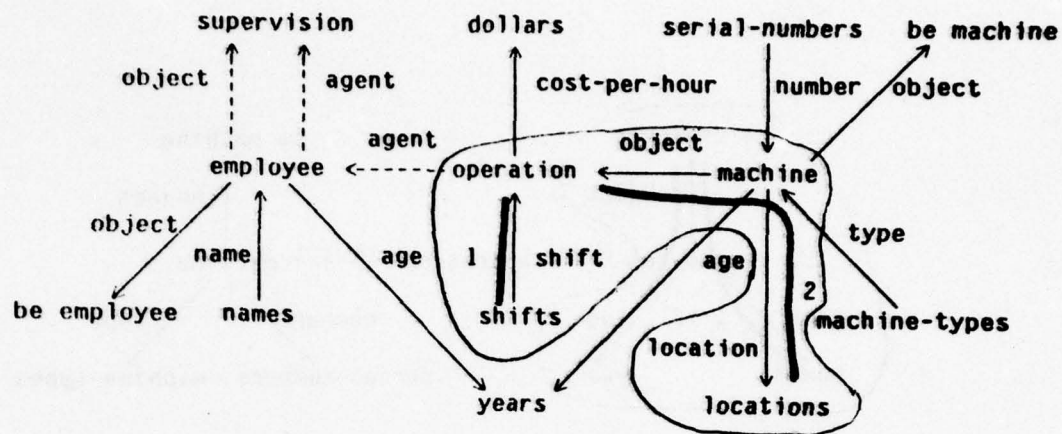


a. Employees Relation

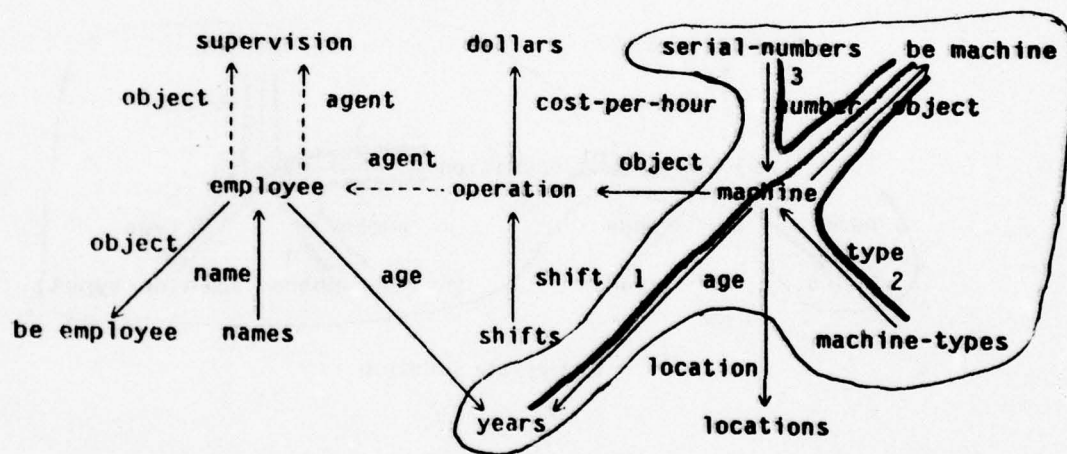


b. Operate Relation

Figure 8.6 - Relational Views Corresponding to Figure 8.5

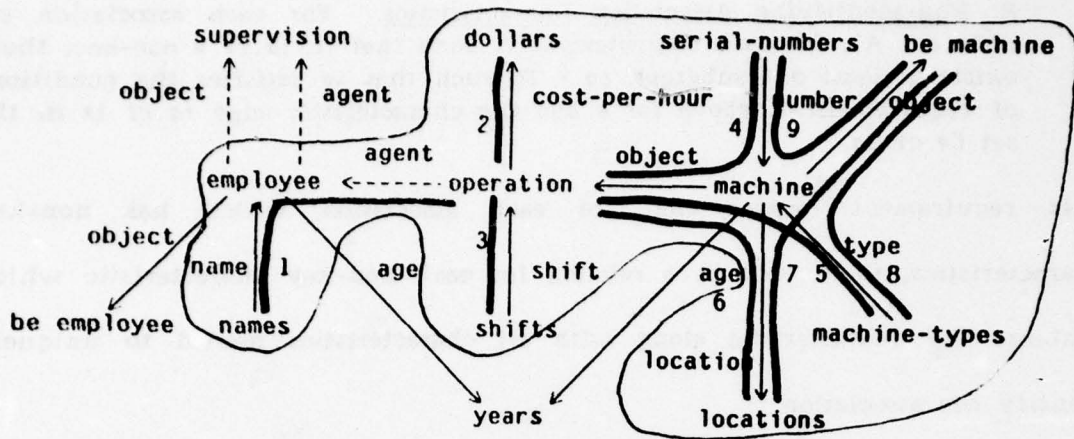


a. Locations-used Relation

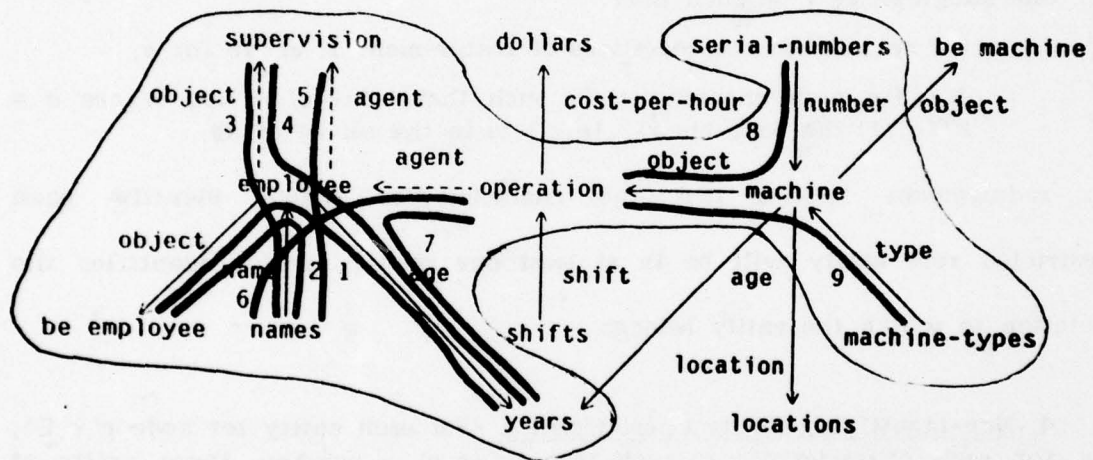


b. Machine-age Relation

Figure 8.7 - Relational Views corresponding to Figure 8.2
(Continued on Next Page)



c. Operations Relation



d. Supervise-operate Relation

Figure 8.7 - Relational Views corresponding to Figure 8.2

2. Non-identifying Association Characteristics. For each association set node $a \in A'$, for each characteristic c , such that $fc'(a,c) = \text{non-key}$, there exists at least one subgraph $sg \in SG$ such that sg satisfies the conditions of requirement 1. above for a and the characteristic edge (a,c) is in the set Ca of sg .

This requirement implies that for each association which has non-key characteristics, there will be a relation for each non-key characteristic which contains the characteristic along with all characteristics needed to uniquely identify the association.

3. Non-identifying roles. For each association set node $a \in A'$, for each role r , such that $fr'(a,r).restrict = \text{unrestricted}$, there exists at least one subgraph $sg \in SG$ such that:

- a. sg satisfies the conditions of requirement 1. above for a .
- b. For each characteristic c such that $fc'(e,c) = \text{key}$ where $e = R'(a,r)$, the path $((a,r), (e,c))$ is in the set Pa of sg .

This requirement implies that the characteristics which identify each unrestricted role entity will be in at least one relation which identifies the association to which the entity belongs.

4. Non-identifying Entity Characteristics. For each entity set node $e \in E'$, for each characteristic c , such that $fc'(e,c) = \text{non-key}$, there exists at least one subgraph sg such that:

- a. The path $((be\ e, \text{object}), (e,c))$ is in sg .
- b. For each characteristic c' such that $fc'(e,c') = \text{key}$, the path $((be\ e, \text{object}), (e,c'))$ is in sg .

This requirement implies that for each entity set type there will be relations which associate each non-key characteristic with the identifying characteristics of the entity.

As a simple example, we will examine how the subgraphs of Figure 8.6 satisfy the four requirements.

1. Existence and Identity. There are three elements of A' , be employee, be machine and operation. be employee has one restricted role object which has one key characteristic name. So path 1 of Figure 8.6.a satisfies this requirement for be employee. Similarly, path 4 of Figure 8.6.b satisfies the requirement for be machine.

operation has the restricted role object with key characteristic number. Path 2 of Figure 8.6.b satisfies this requirement for this association set node.

2. Non-identifying Association Characteristics. None of the association nodes in Figure 8.6 have characteristics, so this requirement is trivially satisfied. (For the schema of Figure 8.7, path 2 in conjunction with operation identifying paths 3, 4 and 5 of Figure 8.7.c satisfies this requirement for the non-key characteristic cost-per-hour of the operation association.)

3. Non-identifying Roles. The key characteristic name of the unrestricted role agent of operation is included in path 1 of Figure 8.6.b. Hence the requirement is satisfied since this subgraph also has path 2 which satisfies requirement 1 for operation.

4. Non-identifying Entity Characteristics. The non-key characteristics age of employee and type of machine are included in path 2 of Figure 8.6.a and path 5 of Figure 8.6.b respectively. This requirement is then satisfied since path 1 ((be employee, object), (employee, name)) is also in Figure 8.6.a and path 4

((be machine, object), (machine, number)) is also in Figure 8.6.b.

III. Define the relational schema components.

A. Define P , S , E , D , C , sd and Str . This step involves the definition of the aforementioned relational schema components and appropriate translation functions between the components and the corresponding graph schema components.

Formally, within the constraints to be stated on the translation functions, the choice of elements for these components are arbitrary. But it is the choice of these translation functions which will determine the ease of semantic interpretation of the relational schema in terms of the graph schema. That is, arbitrary mappings would not be very understandable.

There must be a translation function $T_p: A' \rightarrow P$. This function must be a 1-1 function onto $P - \{be\} \in E$. This maps each association set node onto a corresponding predicate.

$T_s: R' \rightarrow S$ is a 1-1 onto function which maps each role name onto a corresponding semantic case. Note that for the examples of this thesis, T_s has always been chosen to be the identity function.

There must also be 1-1 onto functions T_e , T_d , T_c and T_{str} which map the E' , D' , C' and Str' components of g -schema' onto the corresponding components of r -schema. For notational convenience (and a

suggestion for a "preferred" function), we will leave exact mention of these functions out of the algorithms of this and following chapters assuming the identity function for each of these. They would be included, if desired, in the same manner in which T_s will be used. sd of r -schema must be defined such that domain membership is preserved by T_d and T_{str} .

B. Given a set SG of subgraphs of g -schema', define the relation heading and names schema components, Rel and R -names. For each subgraph $sg \in SG$ define a relational heading as follows:

1. Partition the subgraph corresponding to the grouping of the columns by predicate:case pairs as desired in the relation's heading.

Formally, partition the set of paths Pa of sg such that:

a. For any two paths $((a,r), (e,c))$ and $((a',r'), (e',c'))$ in the same partition:

i. $a = a'$

ii. $e = e'$

iii. $c = c'$ implies there also exists paths $((a,r), (e,c'))$ and $((a',r'), (e',c))$ in the partition.

iv. $C(e,c) = C(e',c')$

b. For any two association set nodes, a and a' , if every partition which contains a path terminating on a also contains a path terminating on a' , then a must be the association be e for some entity set e .

This step defines the column grouping of the relation, i.e. which predicate:case pairs will be grouped together. For example, referring

to the numbered paths in Figure 8.6.b, the Operate relation is defined by the partition $\{\{1\},\{2,3,4,5\}\}$. This specifies that the name characteristic of operate:agent will form one column group while the number and type characteristics for be machine:object and operate:object are grouped together. For the Employees relation of Figure 8.5, the partitioning of the paths of Figure 8.6.a is simply $\{\{1,2\}\}$. There is only a single column group. The edges Ca specifying characteristics of associations will be grouped in step 3 below according to the association described.

Restriction a. above implies that every predicate:case pair in the same column group is defined for the same entity type and the same characteristics. It is also required that two cases of the same predicate cannot be in the same column group.

Requirement b. above says that for any two predicates other than be e, there must be at least one case for each predicate which is in a column group without the other predicate. This allows, through the use of null values, the existence of tuples which state that an association of one type exists while an association of the other type does not exist. be e is an exception because it is mandatory for every entity type.

The partitioning of the paths P_a in Figure 8.7 to define the columns describing predicate:case pairs in Figure 8.2 is as follows:

<u>Relation (Figure 8.2)</u>	<u>Path Figure</u>	<u>Partition</u>
Locations-used	8.7.a	{{2}}
Machine-age	8.7.b	{{1,2,3}}
Operations	8.7.c	{{1},{4,5,6,7,8,9}}
Supervise-Operate	8.7.d	{{5},{1,2,3,4,6,7},{8,9}}

2. Define the relation's heading for columns describing predicate:case pairs. That is, with each element of the partition defined in the step above associate an element of type H (top level grouping) where this element is

$((p,s) \mid (a,r) \text{ is the first element of a}$
 $\text{path in the partition element}$
 $\text{and } (p,s) = (Tp(a), Ts(r)), e)$

where e is the entity set on which all paths of the partition element terminate.

This specifies the set of predicate:case pairs and the entity type which form the top level heading of this relation column group. With each element of H so defined, associate a function cf where the domain of cf is the set of characteristic names which label characteristic edges in the paths of the partition. cf is defined so that it maps each of these characteristic names into the same domain name which labels the terminus of the characteristic edge defined by fc' .

This specifies the various characteristics and their domains which describe the column group's predicate:case pairs. For the partition of Figure 8.6.b, two such elements of H would be defined - one for each element of the partition. They would be $((\text{operate}, \text{agent}), \text{employee})$ and $((\text{be machine}, \text{object}), (\text{operate}, \text{object}), \text{machine})$. The function mapping name into names would be associated with the first element. The function mapping number into serial-numbers and type into machine-types would be associated with the second element.

3. Define the relation's heading for columns describing predicate characteristics. For each association a which has characteristic edges in the set C_a of sg , define an element h of type H such that $h = \text{Tp}(a)$.

Associate with h a Charfunction cf such that the domain of cf is the set of characteristic names occurring in the association's characteristic edges in the partition. And let it map each such characteristic name into the domain name defined for the edge by fc' .

This specifies the characteristics and domains describing the predicates in the relation.

4. Combine the results of steps 2. and 3. Define the relational heading *heading* in the set *Heading* by letting the domain of *heading*

be the union of those elements of H defined in steps 2. and 3. above.
Let *heading* map each such element of H into the function associated with the element as defined in steps 2. and 3. above.

5. Associate a relation name with the heading defined in step 4.

The above steps carried out for each subgraph completely define the relations in the schema. The function mapping the name into its heading defines the Rel component of the relational schema r -schema. The domain of Rel is the R-names component.

Continuing with the examples previously used, these steps would result in the relational headings of Figures 8.5 and 8.2.

End of Algorithm 8.1

Note that the following are steps in Algorithm 8.1 in which there is the possibility of choice:

1. The choice of subgraphs in step II.
2. The choice of translation functions in step III.A.
3. The choice of path partitioning in step III.B.1.
4. The choice of relation names in step III.B.5.

For a given set of choices and resulting relational schema components, Chapter 9 will present an algorithm for constraint definition.

Chapter 9 - The Restricted Semantic Relation Data Model: Constraints

This chapter deals with the constraint component of the semantic relation data model. To show the motivation for the various kinds of constraints, they are introduced in a "structured" manner. In particular, they are introduced in terms of the algorithm defining constraints for the restricted semantic relation data model. In doing so, the form of the constraints for the "unrestricted" semantic relation data model is also specified. While the definitions of the relational state and operations were relatively simple, the complexity of the relational model resides in the constraint definition component of the schema.

The semantic relation constraints are meant to be in the "spirit" of the types of constraints proposed for the syntactic relational model [Codd 70]. Constraints are predicates on the database state (i.e. functions evaluating to true or false). In addition to applying these predicates directly to the database relations, the predicates may also be applied to relations which can be defined in terms of the database relations via a "relational algebra".

It should be emphasized that this approach is taken here not because it was necessarily considered a "good" approach. Rather, it is done to provide a vehicle for studying an application of the general framework for equivalence problems and to explore the characteristics of a semantic version of the relational model. In fact, as will be seen, the kinds of constraints presented here seem to be rather poor when judged on the criterion of understandability

to the user.

9.1 The Semantic Relational Algebra

For the semantic relation data model, the relational algebra is defined by a set of functions which map pairs consisting of a relational heading and a set of tuples into a new relational heading and set of tuples. We call these the *constraint operation types*. The basic difference between these operation types and the Insert-tuples and Delete-tuples operation types is that the constraint operation types create new relational headings. That is, they define a new form for the relational table. Insert-tuples and Delete-tuples, on the other hand, only change the state of the relation - the set of tuples in the relation.

The constraint operations which create new relational headings are used only in specifying constraints on the state of the database. They would not be used by the database system user to create new relations. While this ability to dynamically create new relations would be a worthwhile feature, it presents a data model too complex for the purposes of this thesis. The constraint operation types are Project, Case-join, Predicate-join and Conjunction. The constraint operation types and constraint types are formally defined in Appendix 3 (relying on the definitions in Appendix 2). Each constraint and constraint operation type will be explained in detail as it is needed by the restricted semantic relation data model constraint definition algorithm.

The operations are semantic counterparts of relational operations defined for

syntactic databases as in [Codd 70]. [Chen] points out that unrestrained use of the syntactic operations in [Codd 70] can lead to misunderstandings and ambiguities when trying to give semantic interpretations to the syntactic relations. On the other hand, the operations presented here preserve the semantics of the arguments to the operations.

9.2 Constraint Types

We allow five types of constraints. Each constraint type is a function which evaluates to a Boolean value. The types of constraints are Equal, Equalc, Subset, Non-null and Col-func. These types of constraints are sufficient for the purpose of defining relational application models equivalent to semantic graph application models. It is not intended that they be as "expressive" as any given user might desire. More general forms of constraints have been proposed for syntactic relational models ([Hammer and McLeod] and [Stonebraker]).

It is conjectured, though not proved, that with one exception the constraint types and constraint operation types are all necessary to define constraints such that each semantic relation application model with schema components derived by Algorithm 8.1 is equivalent to the semantic graph application model from which it was derived. The exception is that an instance of the Equalc constraint can be expressed as two instances of the Subset constraint. "Necessary" is meant in the sense that if any one constraint or constraint operation type were not allowed, those remaining would not be

sufficient. This does not rule out some other set of sufficient constraints and operations.

Formally, the set of allowed constraints in the unrestricted semantic relation data model are defined as follows. Those of the restricted semantic relation data model form a subset of the constraints in the unrestricted data model.

Definition 9.1: The constraints in a semantic relation schema must be functions of the form:

```
constraint(rel: [R-names→Heading], state: R-state)
  returns Boolean

  return con-typei(arg1, arg2, . . . argn)
```

where $1 \leq i \leq 5$

```
and for i = 1: con-typei = Equal
    arg1 ∈ (Heading × Relation)
    arg2 ∈ (Heading × Relation)
for i = 2: con-typei = Subset
    arg1 ∈ (Heading × Relation)
    arg2 ∈ (Heading × Relation)
for i = 3: con-typei = Equalc
    arg1 ∈ (Heading × Relation)
    arg2 ∈ (Heading × Relation)
for i = 4: con-typei = Non-null
    arg1 ∈ (Heading × Relation)
for i = 5: con-typei = Col-func
    arg1 ∈ Heading
    arg2 ∈ Heading
    arg3 ∈ (Heading × Relation).
```

For brevity, we will express these constraints in this thesis using the

following notation:

$$\text{con-type}_i[\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n].$$

The correspondence between this notation and the functional form of Definition 9.1 should be clear.

We will also express those $\text{arg}_i \in (\text{Heading} \times \text{Relation})$ in terms of compositions of the constraint operations applied to the relations in the database state. When stating the name r of a database relation in such an expression, we are referring to the pair consisting of the relational heading from the schema ($\text{Rel}(r)$) and the relational state ($\text{R-state}(r)$).

Figure 9.8, located at the end of this chapter, presents the constraints associated with Figure 8.5. Figure 9.9, also located at the end of this chapter, presents the constraints associated with Figure 8.2. It can be seen that a fair number of constraints are needed even for a database with a small number of relations. Each of the constraints in Figure 9.8 will be identified with the step in the algorithm which mandated its definition.

9.3 The Constraint Definition Algorithm

Algorithm 9.1:

Consider a semantic graph schema

$$g\text{-schema} = (E, A, D, r, c, fr, fc, sd, R, C, Str)$$

and a semantic relation schema

$$r\text{-schema} = (Rel, Con, R\text{-names}, P, S, E, D, C, sd, Str).$$

If the semantic relation schema components, except for the constraint component *Con*, were derived from *g-schema* by Algorithm 8.1 and if the *Con* component is derived by the algorithm to follow, then *r-schema* is a schema in the *restricted semantic relation data model*.

Our goal is to define constraints so that database states based on *r-schema* are always equivalent to those based on *g-schema*. Definition 7.1 defines the items of concern for such an equivalence. Accordingly, the outline of this algorithm is:

I. Define base expressions in terms of the database relations so that there is a base relation

- for each entity type identifier
- for each association type identifier
- for each non-identifying characteristic of an association type
- for each non-identifying role of an association type
- for each non-identifying characteristic of an entity type.

A. Identify portions of the semantic graph schema for each base expression.

B. Define the base expressions.

II. Define constraints on those portions of the database identified by the base expressions so that:

A. All alternative choices for base expressions are equal.

B. Each entity has only a single value for each non-identifying characteristic.

C. Each association has a single:

1. Value for each non-identifying characteristic.
2. Entity filling a role for non-identifying roles.

D. Roles are only filled by existing entities for both of:

1. Identifying roles.
2. Non-identifying roles.

E. Mandatory role requirements are fulfilled.

F. All characteristics of entities and associations are assigned values.

III. Define each database relation in terms of base expressions.

A. Build the expression for each relation to include:

1. Desired characteristics of each entity in the relation.
2. Desired characteristics and roles of associations in the relation.
3. The combination of information concerning a single predicate.
4. The combination of different predicates into a single relation.

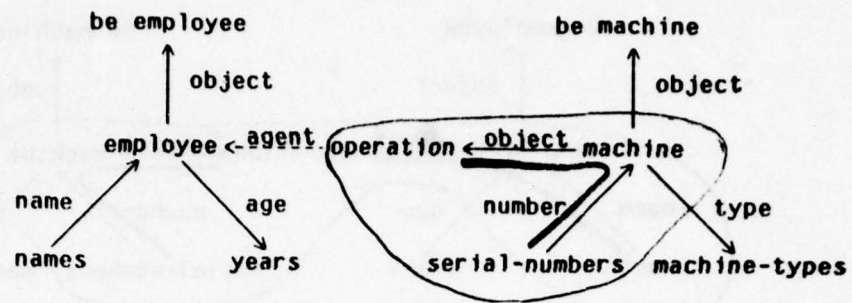
B. Define the constraint enforcing the composition of base relations into the database relation.

The detailed algorithm itself follows.

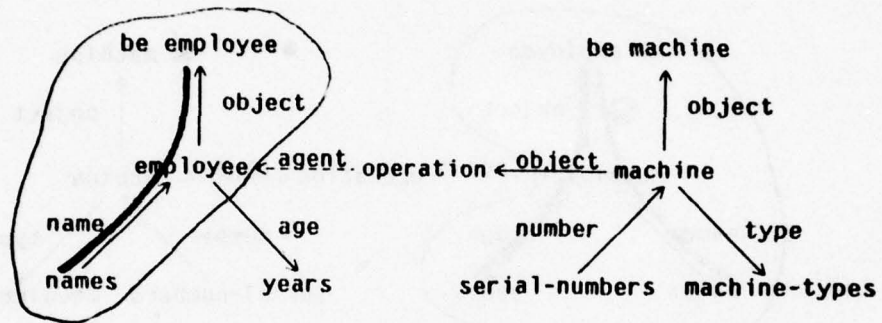
I. The base expressions can be considered to define the "basic facts" about the application. One such base expression is defined for each of the requirements on subgraphs of the augmented semantic graph schema as specified previously in step II of Algorithm 8.1.

A. Requirements 1 to 4 in step II of Algorithm 8.1 state that certain paths and characteristic edges must be present in some subgraph. For each such required set of paths and characteristic edges we define a subgraph of this subgraph. This sub-subgraph would consist of the *minimal* set of paths and edges which satisfy the stated requirement. In the case where there are several such sub-subgraphs, any arbitrarily chosen sub-subgraph is used. (In any case, any such sub-subgraph would specify an identical subgraph of the total augmented semantic graph schema.)

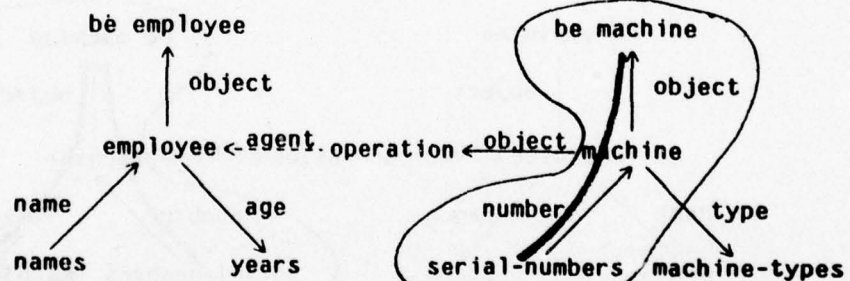
For the relations of Figure 8.5, the sub-subgraphs are shown in Figure 9.1 as subgraphs of the graphs in Figure 8.6 which define the relational "views" of Figure 8.5. The requirements to be satisfied by the sub-subgraphs can be informally restated as:



a.



b.



c.

Figure 9.1 - Sub-subgraphs of Figure 8.6
(Continued on Next Page)

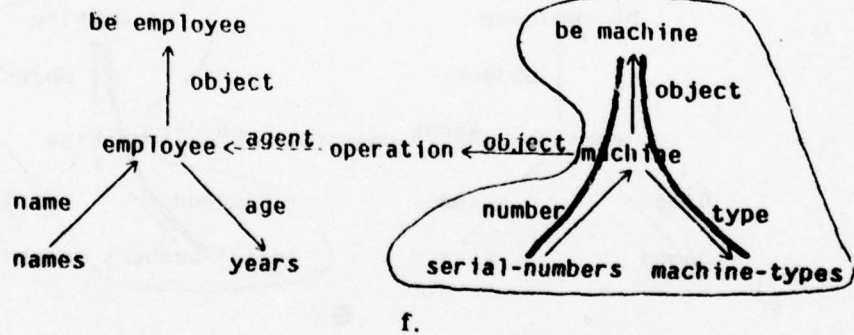
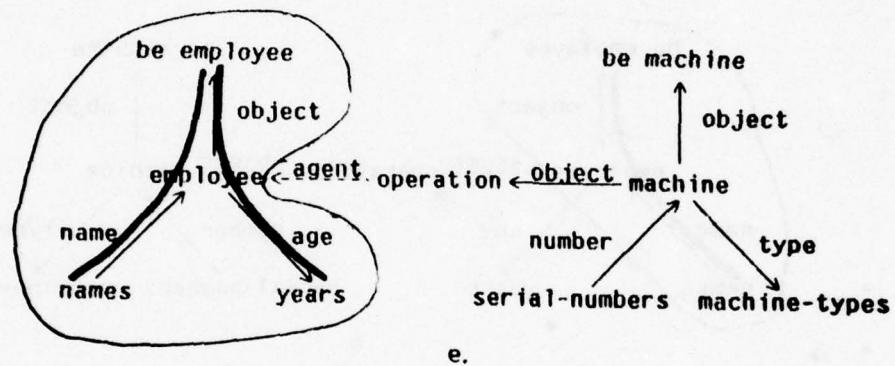
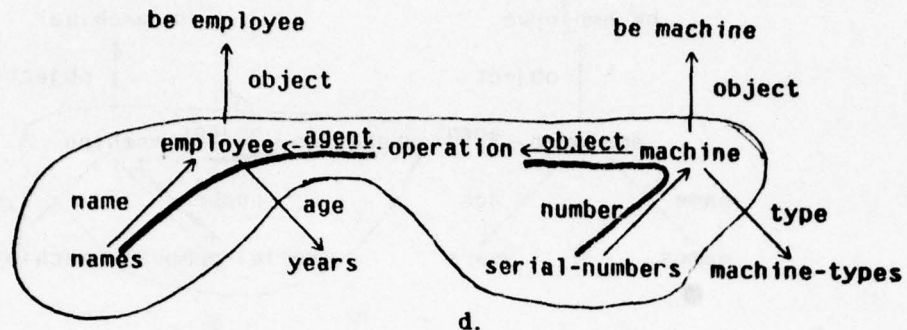


Figure 9.1 - Sub-subgraphs of Figure 8.6

1. Existence and Identity. For each association type, there is a sub-subgraph which contains the minimal set of edges and paths which form the identifying characteristics of the association type.

For the example pertaining to Figure 8.5, the corresponding sub-subgraphs are shown (as subgraphs of Figure 8.4.a) in Figures 9.1.a, b, and c for the association types operation, be employee and be machine, respectively.

2. Non-identifying Association Characteristics. For each non-key characteristic of an association type, there is a sub-subgraph which contains the corresponding characteristic edge and also all of the paths and edges needed to identify the association.

None of the association types in Figure 8.4.a have characteristics, so there are no such sub-subgraphs of Figure 8.4.a.

3. Non-identifying Roles. For each unrestricted entity role in an association type (i.e. those which are not used to identify the association), there is a sub-subgraph which contains the key characteristic paths of this role and also the identifying paths and edges of the association.

For the Figure 8.5 example, Figure 9.1.d shows the sub-subgraph for the only unrestricted entity role, the agent role of the operation association.

4. Non-identifying Entity Characteristics. For each non-key characteristic of an entity type, there is a sub-subgraph which consists of a path from the be *type* association node to each key characteristic edge and a path from the be *type* node to the non-key characteristic node.

Figures 9.1.e and f show the sub-subgraphs for the non-key characteristic age of employee and the non-key characteristic type of machine respectively.

Note that each sub-subgraph shown in Figure 9.1 is contained in one of the subgraphs shown in Figure 8.6.

B. Given the specification of the sub-subgraphs as paths and edges of specified subgraphs, the relational expressions defining the base relations can be specified. This is done by first following step III.B of Algorithm 8.1 for each sub-subgraph to define relational headings. If for *sub-subgraph_i* which is a subgraph of *subgraph_j* corresponding to *Relation_j*, we thusly define *heading_i*, we then define the base relational expression for this sub-subgraph as

$$\text{Baserelation}_i = \text{Project}(\text{heading}_i, \text{Relation}_j).$$

Figure A3.1 defines the Project constraint operation. The operation allows the definition of relations which consist of statements directly implied by the argument relation. This is done by eliminating columns of the relation and/or eliminating certain of the predicate:case pairs from the heading of the relation for those columns which have more than one predicate:case pair.

For example, using the Operations relation from Figure 8.2,

Project(

be machine:object	
machine	
number	type
serial-numbers	machine-types

, Operations)

or in set notation

Project((((be machine,object)),machine),
((number,serial-numbers),(type,machine-types))),
Operations)

results in the Be-machine-base relation shown in Figure 9.2. Note that columns have been eliminated and that the predicate:case pair operate:object has been eliminated from the retained columns. Yet the meaning of the relation is still apparent.

It is assumed that the relational arguments to Project, as well as the other constraint operation types, are in proper form. That is, the tuples are in the form dictated by the accompanying heading. This would always be the case when all arguments are based on the well-formed database relations.

The first part of Figure 9.8 defines the base expressions corresponding to the Figure 8.5 example. Note that the Employee-char-base base

Be-machine-base

be machine:object	
machine	
number	type
serial-numbers	machine-types
NZ745	lathe
JCL181	press

Figure 9.2 - Result of Project Operation

expression is simply identified with the Employees relation of Figure 8.5. This uses the first simplification rule shown in Figure 9.3. Other simplification rules to be used are also in Figure 9.3. The "completeness" of these rules will be discussed later.

1. $\text{Project}(h, (h, r)) = (h, r)$
2. $\text{Case-join}(c, c, \text{Case-join}(c', c, (h_1, r_1), (h_2, r_2)), (h_2, r_2)) = \text{Case-join}(c', c, (h_1, r_1), (h_2, r_2))$
3. If $\text{Project}(h, (h', r_1)) = (h, r_2)$
then $\text{Case-join}(c, c, (h', r_1), (h, r_2)) = (h', r_1)$

where $h, h', h_1, h_2 \in \text{Heading}$
 $c, c' \in \text{Case-col}$
 $r, r_1, r_2 \in \text{Relation}$

Figure 9.3 - Some Equivalence Rules

We have been using the term *expression* to emphasize that base expressions simply identify a portion of the relation in the database state. They do not identify relations which are visible to users of the database system. They are used only to define the semantic constraints on the database state.

However, given this understanding, we can now refer to *base relations*. A base relation specifies the result of evaluating a base expression in a given database state. As in the case of base expressions, base relations are only used to define the relational constraints. They need not actually ever "exist" or "be produced" in the implementation of a relational system.

II. Given the definitions of the base expressions, we then can formally define the set of constraints. Some constraints will define certain properties required of the base relations while others will describe how each database relation can be expressed as some composition of the base relations.

Of course, since the base relations are defined as projections of the database relations, the end result is that all of the constraints in fact specify properties of the database relations actually seen and operated on by the user. The base relations are used as a method of structuring the constraints and will be used in Chapter 10 to define equivalent operations between the semantic graph and relational models.

The following sections define constraints on the base relations, but as just

pointed out, when specified in complete form they define properties of the database relations. Whenever a base relation is specified, the complete expression defining the base relation should be substituted for the base relation name.

Each base expression was defined in step I as a consequence of one of the four requirements of step II in Algorithm 8.1. So we will refer to each base relation as a "type i base relation" where $1 \leq i \leq 4$ and where the types have associated names as follows:

- type 1 - Existence and Identity
- type 2 - Non-identifying Association Characteristics
- type 3 - Non-identifying Roles
- type 4 - Non-identifying Entity Characteristics

A. The first kind of constraint required makes sure that whenever the roles or characteristics of an association type are defined by several base relations, every association instance of that type in the database state described by one such relation is also described by every other relation pertaining to that association type.

In the case of a predicate of the form $be \text{ type}$, we want to make sure that if the entity has non-key characteristics, then every association described by the type 1 base relation for this association type is also described in the type 4 base relations for this association. In the Figure 9.8 example, we want to make sure that every entity identified in base relations Be-employee-base and Be-machine-base have characteristics defined in Employee-char-base and Machine-char-base respectively.

In the case of other than be *type* associations, we want to make sure that the identifying characteristics which occur in its type 1 base relations also occur in any type 2 or type 3 base relations associated with the appropriate association type. In the Figure 9.8 example, this means that the sets of values which occur in the Operation-base relation also occur in the Operation-case-base relation specifying the agent role for each association of type operation.

The formal requirement is as follows:

For each type 1 *Baserelation_i* with *heading_i*, for each type 2, 3 or 4 *Baserelation_j* for the same predicate, there is a constraint

Equal [Project(*heading_i*, *Baserelation_j*), *Baserelation_i*].

The Equal constraint type, defined in Figure A3.2 simply checks that two relations are of equal form and have identical sets of tuples.

In Figure 9.8, the corresponding constraints for this step are C1 - C3. Note that in this case the constraints are superfluous because in each of them they assert the equality of identical projections of the database relations. When the information about a single association is distributed over several relations, this is not the case and the constraints become necessary to preserve the semantic integrity of the database state. Such is the case in Figure 9.9 for constraint C2 where the Machine-char-base-1 and Be-machine-base base relations are defined in

terms of the Machine-age and Operations database relations respectively.

B. We want to make sure that each entity in the database state has associated with it only one value of each non-key characteristic. For each type 4 *BaseRelation_i* with *heading_i*, there is a constraint of the form

$$\text{Col-func } [\{ (h.cf) \}, \{ (h.cf') \}, \text{BaseRelation}_i]$$

where *h* is the top level column grouping for *heading_i* as defined in step III.B.4 of Algorithm 8.1, *cf* maps each key characteristic for the entity type into the appropriate domain name, and *f'* maps the non-key characteristic into its domain.

Col-func enforces constraints similar to the functional dependencies of [Codd 72a]. The Col-func constraint type, defined in Figure A3.3 requires that for a group of columns, whenever two tuples in a relation have equal values for all characteristics in these columns, the values for another specified set of columns must also be equal. This constraint is used in maintaining in the relational state the notions of key characteristics and mandatory cases.

Constraints C4 and C5 in Figure 9.8 are defined as a result of this step.

C. Similar to the constraints of B., we want to make sure that every association described in the database state has associated with it only one value of each non-key characteristic and only one entity for each unrestricted role.

1. For each type 2 *Baserelation_i* with *heading_i*, there is a constraint of the form

Col-func [*heading'*, (*p*, *cf*), *Baserelation_i*]

where *p* is the predicate described in *Baserelation_i*, *cf* maps the non-key characteristic into its domain, and *heading'* is *heading_i* exclusive of (*p*, *c*).

Since none of the associations in Figure 8.6 have characteristics, there are no constraints in Figure 9.8 created by this step. However, constraint C9 in Figure 9.9 is such a constraint.

2. For each type 3 *Baserelation_i* with *heading_i*, there is a constraint of the form

Col-func [*heading1*, *heading2*, *Baserelation_i*]

where *heading2* defines all characteristics in *Baserelation_i* describing the unrestricted case and *heading1* is *heading_i* exclusive of *heading2*.

Constraint C6 in Figure 9.8 is the appropriate constraint for the unrestricted operate:agent case.

D. For each entity of a given type in the semantic graph state, there must be a corresponding tuple in each base relation describing the *be type* predicate. We want to make sure that the key characteristics in cases of other predicates identify existing entities.

1. For each type 1 *BaseRelation_i* not specifying a *be type* predicate, there is a constraint for each case in the relation of the form

Subset [Project((({*p,s*}), *e*), *cf*), *BaseRelation_i*), *BaseRelation_j*]

where *p* is the predicate described in this *BaseRelation_i*, *s* is the case of concern, *e* is the entity type, *cf* is the function specifying the key characteristics of the entity type and *BaseRelation_j* is the type 1 base relation defined for the *be e* predicate.

The Subset constraint type, defined in Figure A3.4 checks that for two relations defined for a single predicate:case pair, the characteristic values in the first relation specified are a subset of the characteristic values in the second relation specified. Subset does not check that tuples in the first relation are a subset of the tuples in the second relation. Rather it looks only at the characteristics and their values. This is because it is desired to compare characteristics describing different predicate:case pairs.

For example, constraint C7 in Figure 9.8 is such a constraint stating that every machine number which occurs in the Operate-base relation also occurs in the Be-machine-base relation. But note that this compares the set of tuples describing the operate:object pair to the set of tuples for be machine:object.

2. For each type 3 *Baserelation_i*, there is a constraint of the form

Subset [Project((({*p*,*s*}), *e*), *cf*), *Baserelation_i*), *Baserelation_j*]

where *s* is the unrestricted case defined in this relation and *p*, *e*, *cf* and *Baserelation_j* are as in 1 above.

This step results in constraint C8 of Figure 9.8 specifying that every employee name Operate-case-base also occurs in the Be-employee-base relation.

E. In the semantic graph schema, it may happen that a particular role in some association type is specified as mandatory for some entity type. When this happens, we must make sure that the relations reflect this feature of the application.

For each entity type $e \in E'$, for each association $a \in A'$, and role *r*, such that $fr'(a,r).mand = \text{mandatory}$ and $R'(a,r)=e$, define a constraint

Equalc [Project(*heading*, *Baserelation_i*), *Baserelation_j*]

where *Baserelation_i* is the type 1 relation if *r* is restricted or else the type 3 relation if *r* is unrestricted, *heading* is the column group in *Baserelation_i* concerned with the predicate:case pair $\uparrow p(a) : \uparrow s(r)$, and *Baserelation_j* is the type 1 base relation for the be *e* predicate.

The Equalc constraint type, defined in Figure A3.5 checks that for two relations defined for a single predicate:case pair, the characteristic values in the first relation specified are equal to the characteristic values in the second relation specified.

The *Equalc* constraint type, like *Subset*, compares only the characteristic values. As in *Subset* this is for the purpose of comparing columns describing different predicate:case pairs. This is in contrast to the *Equal* constraint type which requires both values and headings to be identical. An instance of an *Equalc* constraint could be expressed with two instances of the *Subset* constraint. *Equalc* is retained for clarity.

Constraint C9 in Figure 9.8 states that every machine must be the object of an operate predicate in the *Operate-base* relation.

F. For each *Baserelement*_i, we define the constraint

Non-null [*Baserelement*_i].

The *Non-null* constraint defined in Figure A3.6 requires that the null value cannot occur anywhere in a specified relation.

This simply states that for every entity in the database state, each characteristic for its entity type is defined and for each association, every role and characteristic for its association type is defined. This results in constraints C10-C15 of Figure 9.8.

This ends the specification of constraints on the base relations.

III. So far we have defined the base relations and their constraints. Now we must complete the definitions of the constraints by specifying how each database relation is composed out of the base relations. *Since each base relation is the projection of some database relation, this has the effect of specifying conditions which must hold within a single database relation or between several database relations.*

If each database relation were identical to a single base relation, this last set of constraints would not be necessary. These constraints specify how the various "basic" facts are combined to form any view of the application a user wishes to see in a single relation. These constraints are somewhat similar to the definition of *virtual relations* in syntactic relational database systems ([Codd 74] and [Chamberlin et. al.]). However the concept of virtual relations is closely tied to the implementation issues in a database system. Our use of base relations is purely for definitional reasons.

There will be one such constraint for each relation in the database. This constraint expresses the equality of the relation with a relation created using the Conjunction, Predicate-join, Case-join and Project operations on the base relations. The Conjunction operation is used to combine relations with statements concerning different predicates into a single relation. Predicate-join combines relations describing different cases or characteristics of a single predicate into a single relation. Case-join combines relations which specify different characteristics of an entity

type into a single relation. Project eliminates from a relation those columns or predicates which are not of interest.

[Codd 70] defines the join operation to combine two syntactic relations into a single relation. The use of this operation is criticized by [Chen] as it may allow the creation of a relation which has no semantic interpretation in the application being described in the database. We solve this problem by defining three different operations to form a semantic relation from two other semantic relations.

The procedure for defining the constraints is to do the following for each $Relation_i$ with $heading_i$ defined in r-schema.

A. Build the expression for each relation.

1. Include the desired characteristics of each entity in the relation. For each $predicate_j:case_k$ pair occurring in $heading_i$ which has non-key characteristics specified by $heading_i$, define the following expression

Case-join(((be e,object)),e),(((predicate_j,case_k)),e),
(be-e-head_n,Be-e-char-base_n),

Case-join(((be e,object)),e),(((predicate_j,case_k)),e),
(be-e-head_{n-1},Be-e-char-base_{n-1}),

.

.

.

Case-join(((be e,object)),e),(((predicate_j,case_k)),e),
(be-e-head₂,Be-e-char-base₂),

Case-join(((be e,object)),e),(((predicate_j,case_k)),e),
(be-e-head₁,Be-e-char-base₁),
(pred-head,Predicate_j-base))))))

where e is the entity type associated with $predicate_j:case_k$ by $heading_i$. $Be-e-char-base_1$ to $Be-e-char-base_n$ and $be-e-head_1$ to $be-e-head_n$ are the type 4 base relations and headings for the n non-key characteristics of entity type e mentioned in the $predicate_j:case_k$ columns of $heading_i$. $Predicate_j-base$ and $pred-head$ are the base relation and heading defined for $predicate_j$. This base relation is of type 1 if $case_k$ is restricted and of type 3 if $case_k$ is unrestricted.

This defines a relation consisting of all of the key characteristics of the entity type along with those non-key characteristics occurring in the relation for the specified predicate:case pair.

The Case-join function is defined in Figure A3.7. This function combines the characteristics of an entity type occurring in two different relations into a new relation. Tuples are matched according

to the equality of the common characteristics defined in the relations for the entity type and predicate:case of concern.

For example, using the Employee and Operate relations of Figure 8.5,

Case-join(((be employee,object)),employee),
 (({operate,agent},employee),
 Employee,
 Operate)

results in the Operations relation of Figure 9.4. The information concerning the employees' ages from the Employee relation was added to the Operate relation.

Operations

operate:agent		be machine:object operate:object	
employee		machine	
name	age	number	type
names	years	serial-numbers	machine-types
T.Manhart	32	NZ745	lathe
C.Gershag	40	JCL181	press

Figure 9.4 - Result of Case-join Operation

If, for a tuple in either of the relations being joined, there is no tuple in the other relation with equal values for the common

characteristics, then the information from the unmatched tuple will not appear in the resulting relation. To assure that this operation does not result in false statements concerning the application, this algorithm will only use the operation when all key characteristics of the entity type of concern are present in both relations.

2. Include the desired characteristics and roles of associations in the relation. For each *predicate_j* occurring in *heading_i* which has two or more unrestricted cases or non-key characteristics in *heading_i*, define the following expression

Predicate-join(*predicate_j*, (*pred-head_n*, *Predicate_j-char-base_n*),
Predicate-join(*predicate_j*, (*pred-head_{n-1}*, *Predicate_j-char-base_{n-1}*),
.
.
.
Predicate-join(*predicate_j*, (*pred-head₃*, *Predicate_j-char-base₃*),
Predicate-join(*predicate_j*, (*pred-head₂*, *Predicate_j-char-base₂*),
(*pred-head₁*, *Predicate_j-char-base₁*)))....))

where *pred-head₁* to *pred-head_n* and *Predicate_j-char-base₁* to *Predicate_j-char-base_n* are the type 2 headings and relations for each non-key characteristic and the type 3 headings and relations defined for each unrestricted case.

If *predicate_j* has no unrestricted cases or non-key characteristics in *heading_i*, then the expression defined by this step is the type 1 base relation for *predicate_j*.

If *predicate_j* has one non-key characteristic or unrestricted, then the result of this step is the type 3 or type 2 base relation for the predicate.

This defines a relation which contains each desired unrestricted case or non-key characteristic for the predicate along with the identifying characteristics (restricted case and key characteristics) for

the corresponding association type.

Predicate-join, defined in Figure A3.8 matches information concerning the desired predicate according to those cases and characteristics of this predicate which are common to both relations. For the Operate-case-base and Operate-char-base relations defined in Figure 9.5, the Operate-join relation in Figure 9.5 is the result of

Predicate-join(operate, Operate-case-base, Operate-char-base).

Note that both the agent case and the cost-per-hour characteristic of the operate predicate appear in Operate-join.

If, for a tuple in either of the relations being joined, there is no tuple in the other relation with equal values for the common predicate characteristics and cases, then the information from the unmatched tuple will not appear in the resulting relation. To assure that this operation does not result in false statements concerning the application, this algorithm will only use the operation when all key characteristics of the predicate type of concern and when all key characteristics of the restricted roles are present in both relations.

Operate-case-base

operate:agent	operate:object		operate
employee	machine		
name	number	type	shift
names	serial-numbers	machine-types	shifts
T.Manhart	NZ745	lathe	1
B.Belgim	NZ745	lathe	2
C.Gershag	JCL181	press	1

Operate-char-base

operate:object		operate	
machine			
number	type	shift	cost-per-hour
serial-numbers	machine-types	shifts	dollars
NZ745	lathe	1	40
NZ745	lathe	2	50
JCL181	press	1	75

Operate-join

operate:agent	operate:object		operate	
employee	machine			
name	number	type	shift	cost-per-hour
names	serial-numbers	machine-types	shifts	dollars
T.Manhart	NZ745	lathe	1	40
B.Belgim	NZ745	lathe	2	50
C.Gershag	JCL181	press	1	75

Figure 9.5 - Predicate-join Example

3. Include the combination of information concerning a single predicate. For each $predicate_j$ occurring in $heading_i$ which had an expression defined in step 1. above, define the following expression

$$\text{Case-join}(\{((predicate_j, case_n)), e), \{((predicate_j, case_n)), e), \\ exp_n,$$

$$\text{Case-join}(\{((predicate_j, case_{n-1})), e), \{((predicate_j, case_{n-1})), e), \\ exp_{n-1},$$

.

$$\text{Case-join}(\{((predicate_j, case_2)), e), \{((predicate_j, case_2)), e), \\ exp_2,$$

$$\text{Case-join}(\{((predicate_j, case_1)), e), \{((predicate_j, case_1)), e), \\ exp_1, exp_0)) \dots)$$

where exp_0 is the relational expression defined in step 2 for $predicate_j$ and exp_i is the expression defined in step 1 for $case_i$.

If no expressions were defined for $predicate_j$ in step 1, then the result of this step is exp_0 .

This adds to each column group the desired non-key characteristic for the entity type of concern.

4. Combine the different predicates into a single relation. If *heading_i* contains more than one predicate, then define the following expression

Conjunction(*col-list-1_n*,*col-list-2_n*,
exp_n,

Conjunction(*col-list-1_{n-1}*,*col-list-2_{n-1}*,
exp_{n-1},

.

.

.

.

Conjunction(*col-list-1₂*,*col-list-2₂*,
exp₂,

Conjunction(*col-list-1₁*,*col-list-2₁*,
exp₁, *exp₀*))...))

where *exp_j* is the relational expression defined in step 3 for each *predicate_j*. Each *col-list-1_j* is a sequence of column headings specifying those cases and their entity types of *predicate_j* which occur in *heading_i* with other predicate:case pairs. Each *col-list-2_j* lists column headings consisting of the entity type and predicate:case pairs defining column groups in *exp_j*. Corresponding columns specified by the two lists will be combined.

If there is only one predicate in *heading_i*, then the result of this step is the expression defined in step 3.

This defines a single relation which contains information concerning all of the predicates which are to be in the corresponding database relation.

Conjunction is defined in Figure A3.9. The arguments to the function specify two relations and two lists of column groups. Corresponding entries in the two lists must define predicate:case pairs defined on the same entity type and the same characteristics. It is also required that no single predicate appears in both relations. If this were the case, semantic ambiguity might result. Unlike the situation

with Case-join and Predicate-join, the result of a Conjunction operation contains all information present in both arguments to the operation.

The heading produced by the function is formed by combining the specified column groups and retaining all of the other columns in both relations. The tuples in the resulting relation are formed by combining tuples from the two relations which have identical values in the merged columns and by adding null values to all other tuples.

For example,

```
Conjunction({(((supervise,object)),employee)),
             {(((operate,agent)),employee)),
             Supervise-base,
             Operators)
```

where relations Supervise-base and Operators are defined in Figure 9.6 results in the Jobs relation of Figure 9.6. Note that the null value was introduced since B.Belgim has no supervisor.

B. Finally define the constraint

$$\text{Equal}[\text{Relation}_i, \text{Project}(\text{heading}_i, \text{exp}_0)]$$

where exp_0 is the relation expression defined in step 4.

The Project operation in this constraint eliminates any of those columns used in the Case-join and Predicate-join operations which do not appear in the final heading_i . Such columns would be key characteristics of restricted cases or key characteristics of predicates.

For the relations of Figure 8.5, these steps result in the constraints

Supervise-base

supervise:agent	supervise:object
employee	employee
name	name
names	names
G.Wayshum G.Wayshum	T.Manhart C.Gershag

Operators

operate:agent	operate:object	
employee	machine	
name	number	type
names	serial-numbers	machine-types
B.Belgim T.Manhart C.Gershag	NZ745 NZ745 JCL181	lathe lathe press

Jobs

supervise:agent	supervise:object operate:agent	operate:object	
employee	employee	machine	
name	name	number	type
names	names	serial-numbers	machine-types
--- G.Wayshum G.Wayshum	B.Belgim T.Manhart C.Gershag	NZ745 NZ745 JCL181	lathe lathe press

Figure 9.6 - Example of Conjunction Operation

C16' and C17' of Figure 9.10 at the end of the chapter. The constraints C16 and C17 of Figure 9.8 are derived from C16' and C17' by use of the simple equivalence rules shown in Figure 9.3.

This completes the formal description of the constraint definition procedure. The use of the stated procedure and the three simplifying rules result in the constraints shown in Figure 9.9 for the relational schema shown in Figure 8.2. These constraints include the various types of constraint operations not shown in Figure 9.8. The constraints in these figures should be considered in detail to get a feeling of how they reflect the features of the graph schema.

End of Algorithm 9.1.

The restricted semantic relation data model is then formally defined.

Definition 9.2: The restricted semantic relation data model consists of the set of all semantic relation application models with schemas derivable by Algorithm 9.1.

AD-A066 386

MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/2
EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS FOR DATABASE SYS--ETC(U)
JAN 79 S A BORKIN N00014-75-C-0661
MIT/LCS/TR-206 NL

UNCLASSIFIED

3 OF 4
ADA
066386



9.4 Discussion

While the constraints defined by Algorithm 9.1 are specified as compositions of syntactic expressions in a canonical form, it is important to remember that it is the function represented by the expressions which define the constraints in the schema. Consequently there could be many other ways of expressing each constraint or combination of constraints. The rules in Figure 9.3 are only a few simple equivalences. There are many other such equivalences. [Sagiv and Yannakakis] show that the determination of equivalence for syntactic relational algebra expressions involving only projection and join is NP-complete. The problem is certainly no simpler in the case of the semantic relation operations.

We will now again describe some of the ways in which relational application models can be defined which have no equivalent graph application model. It is possible that either the constraints are not strict enough, allowing relational states with no equivalent valid graph state, or the constraints may be too strict, which would result in excluding from the set of valid relational states that state which is equivalent to some valid graph state.

If constraint C3 was eliminated from Figure 9.9, then the constraints are not strict enough since this would allow statements specifying a machine's age without specifying its location. The graph model, however, requires that all characteristics be defined.

In Chapter 8 an example was presented of over-constraining the database

state. Using the formal notation which has now been introduced, we express the constraint from this example as Figure 9.7. When applied to the relations of Figure 8.2, this constraint says that the types of every machine operated by the employees of a given supervisor must be equal. As discussed in Chapter 8, such a constraint is not expressible in the semantic graph data model. In effect, this constraint specifies properties which must be satisfied by nodes in the semantic graph state which are not connected by an edge. The semantic graph operations do not enforce such constraints.

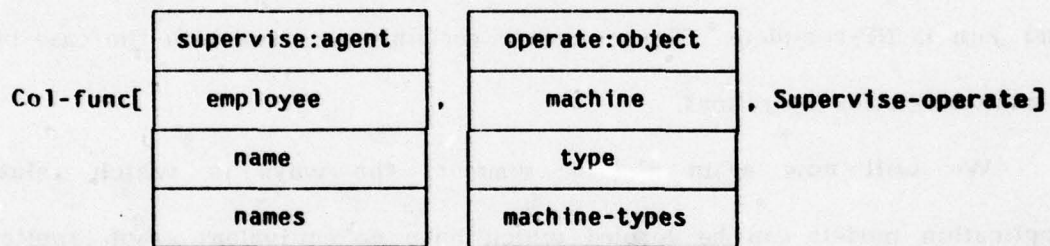


Figure 9.7 - Constraint Causing Non-equivalence

We conclude this chapter with a summary of the main steps in the derivation of restricted semantic relation schemas. Chapter 10 then looks at the problem of the equivalence of the semantic graph and semantic relation data models.

Algorithm 8.1:

- I. Define an *augmented graph schema*.
- II. Select subgraphs of the augmented schema defining that portion of the application which will be described in each relation, making sure that four types of requirements are satisfied.
- III. Define the relational headings and other schema components except for the constraints.

Algorithm 9.1:

- I. Define base expressions as projections of the database relations so that there is a base expression
 - for each entity type
 - for each association type
 - for each non-key characteristic of an association type
 - for each unrestricted role of an association type
 - for each non-key characteristic of an entity type.
- II. Define constraints on the base relations so that
 - all appropriate characteristics are defined for each entity and each association
 - all appropriate roles are defined for each association
 - the key, restricted and mandatory features are enforced.
- III. Define constraints showing how each database relation is related to the set of base relations.

Base Expressions:

Be-machine-base = Project(

be machine:object
machine
number
serial-numbers

, Operate)

Be-employee-base = Project(

be employee:object
employee
name
names

, Employees)

Operate-base = Project(

operate:object
machine
number
serial-numbers

, Operate)

Operate-case-base = Project(

operate:agent	operate:object
employee	machine
name	number
names	serial-numbers

, Operate)

Employee-char-base = Employees

Figure 9.8 - Constraints for Figure 8.5
(Page 1 of 4)

Machine-char-base = Project(be machine:object		, Operate)
	machine		
	number	type	
	serial-numbers	machine-types	

Constraints

C1. Equal[Project(be employee:object	, Employee-char-base), Be-employee-base]
	employee	
	name	
	names	

C2. Equal[Project(be machine:object	, Machine-char-base), Be-machine-base]
	machine	
	number	
	serial-numbers	

C3. Equal[Project(operate:object	, Operate-case-base), Operate-base]
	machine	
	number	
	serial-numbers	

Figure 9.8 - Constraints for Figure 8.5
(Page 2 of 4)

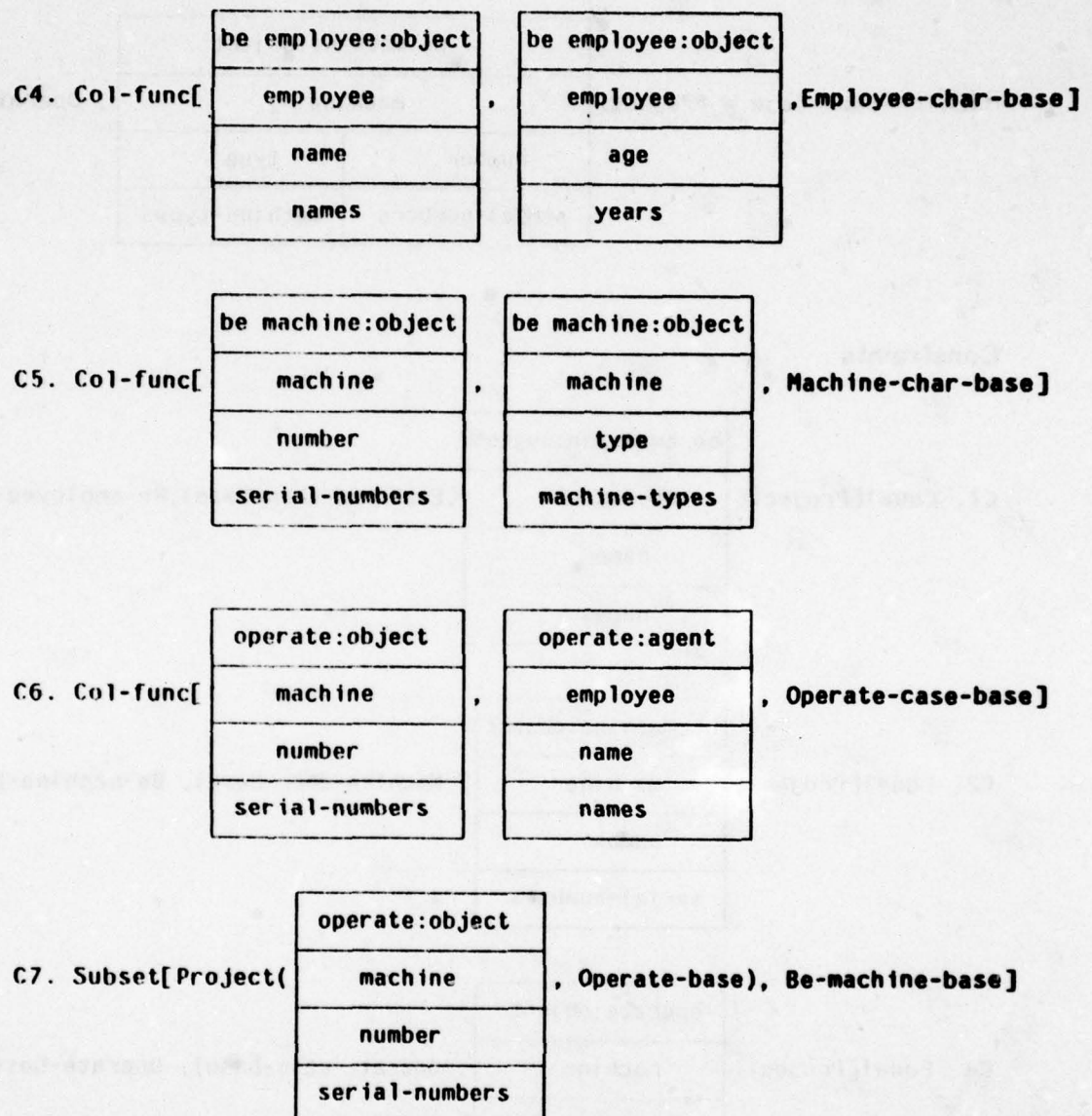


Figure 9.8 - Constraints for Figure 8.5
(Page 3 of 4)

C8. Subset[Project(

operate:agent
employee
name
names

, Operate-case-base), Be-employee-base]

C9. Equalc[Project(

operate:object
machine
number
serial-numbers

, Operate-base), Be-machine-base]

- C10. Non-null[Be-machine-base]
 C11. Non-null[Be-employee-base]
 C12. Non-null[Operate-base]
 C13. Non-null[Operate-case-base]
 C14. Non-null[Employee-char-base]
 C15. Non-null[Machine-char-base]

C16. Equal[Operate,

Conjunction(

be machine:object
machine

,

operate:object
machine

,

Machine-char-base,

Case-join(

be machine:object
machine

,

operate:object
machine

,

Machine-char-base,
 Operate-case-base))]

C17. Equal[Employees, Employee-char-base]

Figure 9.8 - Constraints for Figure 8.5
 (Page 4 of 4)

Base Expressions:

Be-machine-base = Project(

be machine:object	
machine	
number	type
serial-numbers	machine-types

, Operations)

Be-employee-base = Project(

be employee:object
employee
name
names

, Supervise-operate)

Operate-base = Project(

operate:object		operate
machine		
number	type	shift
serial-numbers	machine-types	shifts

Operations)

Supervise-base = Project(

supervise:agent	supervise:object
employee	employee
name	name
names	names

Supervise-operate)

Figure 9.9 - Constraints for Figure 8.2
(Page 1 of 9)

Employee-char-base = Project(be employee:object		, Supervise-operate,
	employee		
	name	age	
	names	years	

Machine-char-base-1 = Machine-age

Machine-char-base-2 = Project(

be machine:object		
machine		
number	type	location
serial-numbers	machine-types	locations

Operations)

Operate-char-base =

Project(operate:object		operate	
	machine			
	number	type	shift	cost-per-hour
	serial-numbers	machine-types	shifts	dollars

Operations)

Figure 9.9 - Constraints for Figure 8.2
(Page 2 of 9)

Operate-case-base =

Project(operate:object		operate:agent	operate
	machine		employee	
	number	type	name	shift
	serial-numbers	machine-types	names	shifts

Operations)

Constraints

C1. Equal[Project(be employee:object	,Employee-char-base),Be-employee-base]
	employee	
	name	
	names	

C2. Equal[Project(be machine:object		,Machine-char-base-1), Be-machine-base]
	machine		
	number	type	
	serial-numbers	machine-types	

Figure 9.9 - Constraints for Figure 8.2
(Page 3 of 9)

C3. Equal[Project(

be machine:object	
machine	
number	type
serial-numbers	machine-types

Be-machine-base]

,Machine-char-base-2),

C4. Equal[Project(

operate:object		operate
machine		
number	type	shift
serial-numbers	machine-types	shifts

Operate-char-base),

Operate-base]

C5. Equal[Project(

operate:object		operate
machine		
number	type	shift
serial-numbers	machine-types	shifts

Operate-case-base),

Operate-base]

C6. Col-func[

be employee:object	,	be employee:object	,Employee-char-base]
employee		employee	
name		age	
names		years	

Figure 9.9 - Constraints for Figure 8.2
(Page 4 of 9)

C7. Col-func[be machine:object		,	be machine:object		,
	machine			machine		
	number	type		age		
	serial-numbers	machine-types		years		

Machine-char-base-1]

C8. Col-func[be machine:object		,	be machine:object		,
	machine			machine		
	number	type		location		
	serial-numbers	machine-types		locations		

Machine-char-base-2]

C9. Col-func[operate:object		operate	operate
	machine			
	number	type	shift	cost-per-hour
	serial-numbers	machine-types	shifts	dollars

Operate-char-base]

C10. Col-func[operate:object		operate	operate:agent
	machine			employee
	number	type	shift	name
	serial-numbers	machine-types	shifts	names

Operate-case-base]

Figure 9.9 - Constraints for Figure 8.2
(Page 5 of 9)

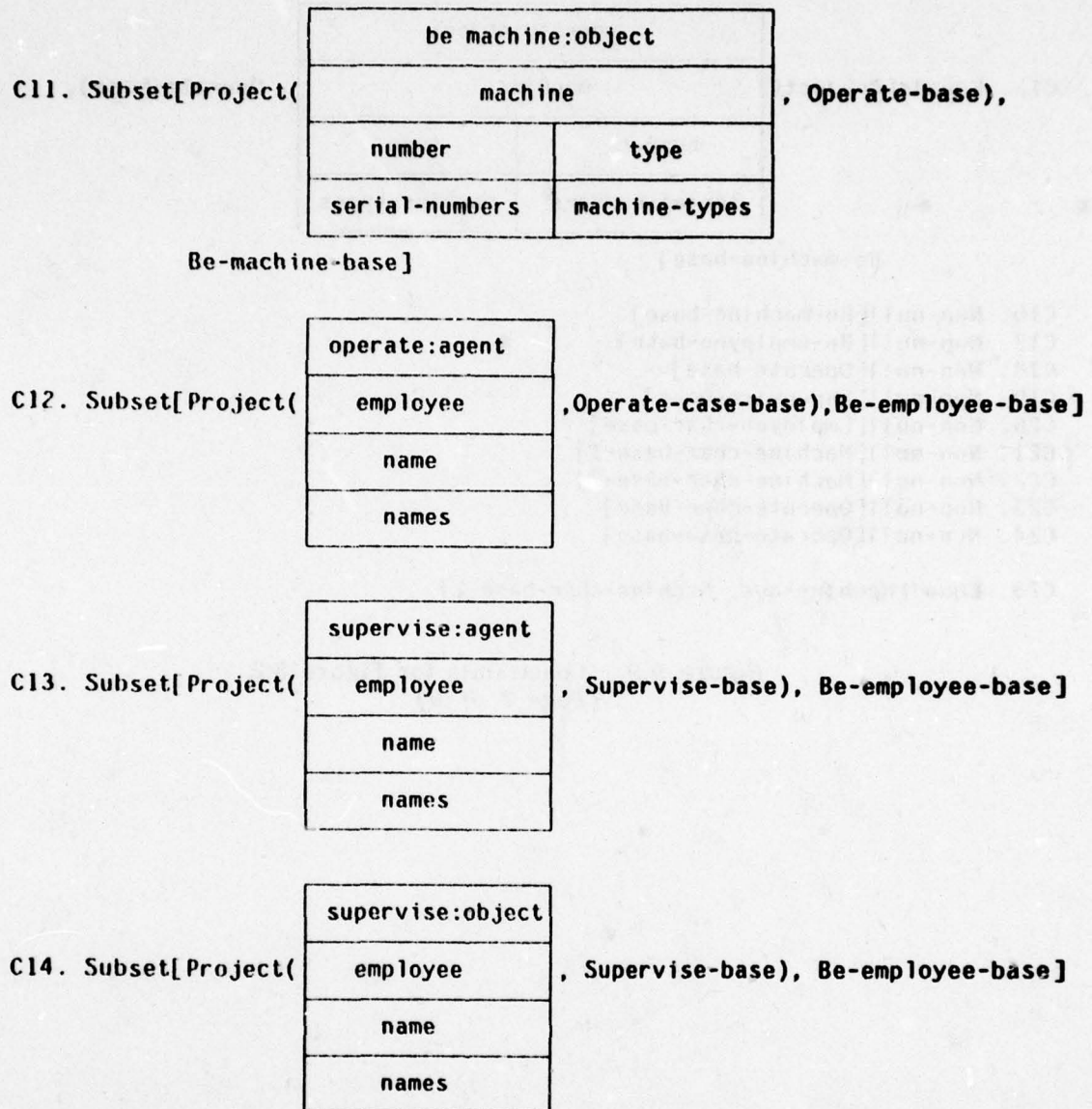


Figure 9.9 - Constraints for Figure 8.2
(Page 6 of 9)

C15. Equalc[Project(operate:object		, Operate-base),
	machine		
	number	type	
	serial-numbers	machine-types	

Be-machine-base]

- C16. Non-null[Be-machine-base]
- C17. Non-null[Be-employee-base]
- C18. Non-null[Operate-base]
- C19. Non-null[Supervise-base]
- C20. Non-null[Employee-char-base]
- C21. Non-null[Machine-char-base-1]
- C22. Non-null[Machine-char-base-2]
- C23. Non-null[Operate-char-base]
- C24. Non-null[Operate-case-base]

C25. Equal[Machine-age, Machine-char-base-1]

*Figure 9.9 - Constraints for Figure 8.2
(Page 7 of 9)*

C26. Equal[Locations-used,

Project(operate:object	operate
	machine	
	location	shift
	locations	shifts

Case-join(be machine:object	operate:object
	machine	

Machine-char-base-2,
Operate-base))]

C27. Equal[Operations,

Conjunction(be machine:object	operate:object
	machine	

Machine-char-base-2,

Case-join(operate:object	operate:object
	machine	

Case-join(be machine:object	operate:object
	machine	

Machine-char-base-2,
Operate-base),

Predicate-join(operate,
Operate-case-base,
Operate-char-base)))]

Figure 9.9 - Constraints for Figure 8.2
(Page 8 of 9)

C28. Equal[Supervise-operate,

Project(supervise:agent	be employee:object supervise:object operate:agent		operate:object	
	employee	employee		machine	
	name	name	age	number	type
	names	names	years	serial-numbers	machine-types

Conjunction(be employee:object	supervise:object operate:agent
	employee	employee

Employee-char-base,

Conjunction(supervise:object	operate:agent
	employee	employee

Case-join(be employee:object	supervise:object
	employee	employee

Employee-char-base,
Supervise-base),

Case-join(be employee:object	operate:agent
	employee	employee

Employee-char-base,
Operate-case-base))))]

Figure 9.9 - Constraints for Figure 8.2
(Page 9 of 9)

C16'. Equal[Operate,

Project(operate:agent		operate:object	
	employee		machine	
	name	number	type	
	names	serial-numbers	machine-types	

Conjunction(be machine:object	operate:object
	machine	machine

Case-join(be machine:object	be machine:object
	machine	machine

Case-join(be machine:object	be machine:object
	machine	machine

Machine-char-base,
Be-machine-base),
Be-machine-base),

Case-join(operate:object	operate:object
	machine	machine

Case-join(be machine:object	operate:object
	machine	machine

Machine-char-base,
Operate-base),
Operate-case-base)))]

Figure 9.10 - Unreduced Constraints of Figure 9.8
(Page 1 of 2)

C17'. Equal[Employees,

Project(be employee:object	
	employee	
	name	age
	names	years

Case-join(be employee:object	,	be employee:object
	employee		employee

Case-join(be employee:object	,	be employee:object
	employee		employee

Employee-char-base,
Be-employee-base),
Be-employee-base))]

Figure 9.10 - Unreduced Constraints of Figure 9.8
(Page 2 of 2)

Chapter 10 - Constructive Proof of Data Model Equivalence

This chapter shows by the construction of equivalent operations that the restricted semantic relation and the semantic graph data models are state dependent equivalent (as defined in Chapter 7). The approach taken is to show the equivalence of any pair of application models consisting of a semantic graph application model and a semantic relation application model whose schema was derived from the graph schema by Algorithms 8.1 and 9.1.

This can be done by induction for each such pair of application models in the following manner:

I. Show that the initial states are equivalent.

II. Show that for any given pair of valid (e.g. reachable) equivalent graph and relational states, and for any given valid semantic graph operation, an equivalent relational operation can be constructed. (That is, the states resulting from the operations applied to the given equivalent states are also equivalent.)

III. Show that for any given pair of valid (e.g. reachable) equivalent graph and relational states, and for any given valid semantic relation operation, an equivalent (possibly composed) graph operation can be constructed.

If this can be shown for every such pair of application models, then it proves that the restricted semantic relation and the semantic graph data models are

state-dependent equivalent. Note that the examples of Chapter 7 show why state-dependent equivalence, as opposed to one of the other forms of equivalence, must be considered.

While a proof to a detailed level would be desirable to show the correctness of an actual implementation based on these constructions of equivalent operations, such a proof would be extremely complicated and lengthy. It would involve proofs concerning every clause of the definitions in Appendices 1, 2 and 3, and every step of Algorithms 8.1 and 9.1. Such a proof would be best done with some sort of mechanized aid. However, we still hope the more informal arguments justifying the claims of this chapter are convincing.

It is important to realize that this approach is not meant to imply the unimportance of detailed formal proofs. On the contrary, it points out the great importance which should be attached to finding workable methodologies for proving program correctness.

10.1 State Equivalence for the Restricted Semantic Relation Data Model

At this point, we return to the problem of defining the equivalence of graph and relational database states. As mentioned in Chapter 7, one approach would be to show that the graph state is a model, in the formal logic sense, for the set of predicate calculus statements associated with the relational state. This approach would require a formal specification of the set of predicate calculus statements associated with a given relation. This approach would be

appropriate for any semantic relation state, not just those in the restricted semantic relation data model.

However, we will take an approach which relies on the canonical form of the constraints in the semantic relation data model. The advantage of this approach is that it is more closely tied to the process of deriving equivalent operations for two given application models.

Equivalence will be determined by comparing the base relations (as defined in Algorithm 9.1) of a relational state with the graph state. This approach is related to the work of [Hall et. al.] which advocates the design of relational databases in terms of "irreducible" or "basic" relations which represent non-redundant "basic facts" concerning the application. The base relations of this thesis also represent such "basic facts". We will use Definition 7.1 to specify the criteria for state equivalence against which we will measure the new definition.

Theorem 10.1: Given a valid (by Definition 6.2) restricted semantic relation database state $r\text{-state}$ in a restricted semantic relation database $((r\text{-schema}, r\text{-state}_0, r\text{-op-types}), r\text{-state})$ and a valid (by Definition 5.2) semantic graph database state $g\text{-state} = (E, AD_r, c)$ in a semantic graph database $((g\text{-schema}, g\text{-state}_0, g\text{-op-types}), g\text{-state})$ where $r\text{-schema}$ is derived from $g\text{-schema}$ by Algorithms 8.1 and 9.1, then $r\text{-state}$ is state equivalent to $g\text{-state}$ if and only if both $g\text{-state}$ and $r\text{-state}$ equal error (signifying that neither represents a possible application state) or when considering the projections of the relations in $r\text{-state}$ which are defined as base relations in the schema derivation, the following are true:

1. Existence and Identity

a. Entities. For each type 1 base relation for a predicate of the form be , the set of tuples in the base relation equals

$$\begin{aligned} \{tuple: \text{Tuple} \mid & (\exists e\text{-id}: E\text{-id})(e\text{-id} \in E \wedge e\text{-id.type} = e \\ & \wedge \text{dom}(tuple) = \{((be\ e, \text{object})), e\} \\ & \wedge tuple(((be\ e, \text{object})), e) = e\text{-id.chars})\} \end{aligned}$$

This means that for every entity in the graph state there is a tuple in the corresponding type 1 base relation consisting of the identifying key characteristics of the entity. Such a tuple specifies the semantic graph characteristic edges for these characteristics.

b. Associations. For each type 1 base relation for a predicate p other than e , the set of tuples in the base relation equals

$$\begin{aligned} \{tuple: Tuple \mid & (\exists a-id: A-id) (a-id \in A \wedge a-id.type = Tp^{-1}(p) \\ & \wedge tuple(Tp^{-1}(p)) = a-id.chars \\ & \wedge dom(a-id.roles) = \\ & \quad \{r: R \mid (((Tp^{-1}(p), Ts(r))), a-id.roles(r).type) \in dom(tuple)) \\ & \wedge (\forall r: R) (r \in dom(a-id.roles) \\ & \quad \supset tuple(((Tp^{-1}(p), Ts(r))), a-id.roles(r).type) = \\ & \quad \quad a-id.roles(r).chars))\} \end{aligned}$$

T_p and T_s are the translation functions defined in Algorithm 8.1.

This means that for each association in the graph state there is a tuple in the corresponding type 1 base relation. This tuple consists of the key characteristics of the association and the key characteristics of the entities in the restricted roles of the association. It therefore specifies those characteristic and role edges.

2. Non-identifying Association Characteristics. For each type 2 base relation describing the non-key characteristic c of a predicate p , the set of tuples in the base relation equals

$$\begin{aligned}
 \{tuple: Tuple \mid & (\exists a-id: A-id) (a-id \in A \wedge a-id.type = Tp^{-1}(p) \\
 & \wedge dom(tuple(Tp^{-1}(p))) = dom(a-id.chars) \cup \{c\} \\
 & \wedge (\forall c': C) (c' \in dom(a-id.chars) \\
 & \quad \supset tuple(Tp^{-1}(p))(c') = a-id.chars(c')) \\
 & \wedge tuple(Tp^{-1}(p))(c) = c(a-id, c) \\
 & \wedge dom(a-id.roles) = \\
 & \quad \{r: R \mid ((Tp^{-1}(p), Ts(r)), a-id.roles(r).type) \in dom(tuple))\} \\
 & \wedge (\forall r: R) (r \in dom(a-id.roles) \\
 & \quad \supset tuple(((Tp^{-1}(p), Ts(r)), a-id.roles(r).type) = \\
 & \quad \quad a-id.roles(r).chars)))
 \end{aligned}$$

This requirement means that for each non-key characteristic of an association in the graph state, there is a tuple containing the characteristics to identify the association along with the same value for the non-key characteristic as specified by the graph state characteristic edge.

3. Non-identifying Roles. For each type 3 base relation for a predicate p describing the unrestricted case s , the set of tuples in the base relation equals

$$\begin{aligned} \{tuple: Tuple \mid & (\exists a-id: A-id) (a-id \in A \wedge a-id.type = Tp^{-1}(p) \\ & \wedge tuple(Tp^{-1}(p)) = a-id.chars \\ & \wedge \{r: R \mid (a-id, r) \in dom(r)\} = \\ & \{r: R \mid (((Tp^{-1}(p), Ts(r))), a-id.roles(r).type) \in dom(tuple)) \\ & \cup \{Ts^{-1}(s)\} \\ & \wedge (\forall r: R) (r \in dom(a-id.roles) \\ & \supset tuple((((Tp^{-1}(p), Ts(r))), a-id.roles(r).type) = \\ & a-id.roles(r).chars))) \} \end{aligned}$$

This requires a tuple for each role edge pertaining to an unrestricted role in the graph state.

4. Non-identifying Entity Characteristics. For each type 4 base relation for the be e predicate describing the non-key characteristic c of the entities of type e , the set of tuples in the base relation equals

$$\begin{aligned} \{tuple: Tuple \mid & (\exists e-id: E-id) (e-id \in E \wedge e-id.type = e \\ & \wedge dom(tuple) = \{((be\ e, object)), e\} \\ & \wedge dom(tuple(((be\ e, object)), e)) = dom(e-id.chars) \cup \{c\} \\ & \wedge (\forall c': C) (c' \in dom(tuple(((be\ e, object)), e)) \\ & \supset tuple(((be\ e, object)), e)(c') = c(e-id, c')) \} \end{aligned}$$

This requires a tuple corresponding to each characteristic edge for an

entity's non-key characteristics.

This ends Theorem 10.1.

Proof of Theorem 10.1:

Now we will prove that Theorem 10.1 fulfills the equivalence requirements of Definition 7.1. We consider each clause of Definition 7.1.

1. For both states, only one value is specified for a single characteristic of an entity or association.

This is true of valid semantic graph database states since by definition there may be only a single characteristic edge of any given type for any given entity or association node. This requirement is guaranteed to be satisfied for valid restricted semantic relation states by the constraints defined in steps II.A, II.B, II.C.1 and II.F of Algorithm 9.1.

The definition of Non-null constraints in II.F guarantees that no null values are assigned. The Equal constraints of II.A guarantee that each base relation has identical sets of values for identifying roles and characteristics. II.B guarantees by Col-func constraints that a unique non-key characteristic is assigned to each entity while II.C.1 guarantees the same for associations.

Since all Case-joins in the constraints of step III include the entity type's key characteristics and since all Predicate-joins in step III include the key characteristics of the association and restricted roles, the necessary conditions are also met in the database relations seen by the user.

2. For both states, only one entity may be specified as filling a given role in an association.

This is true of valid semantic graph database states since by definition there may be only a single role edge of any given type for any association node. This requirement is guaranteed to be satisfied for valid restricted semantic relation states by the constraints defined in steps II.A, II.C.2 and II.D of Algorithm 9.1. II.A requires the equality of the restricted roles in each base relation while II.C.2 requires functionality from the identifiers of an association to the key characteristics for the entity filling an unrestricted role. Furthermore, II.D guarantees by Subset constraints the existence of each entity specified as filling a role.

Requirements 3-6 are satisfied by the existence of the required mappings between the base relation of a semantic relation database state and an equivalent semantic graph state.

3. The states define equal sets of entity and association identifiers.

This is satisfied for equivalent relation and graph states by the Existence and Identity clause of Theorem 10.1.

4. For each entity, the same characteristics are defined in each state and are of equal value.

This is satisfied in Theorem 10.1 by the Existence and Identity clause for key characteristics and the Non-identifying Entity Characteristic clause for non-key characteristics.

5. For each association, the same characteristics are defined in each state and are of equal value.

This is satisfied in Theorem 10.1 by the Existence and Identity clause for key characteristics and the Non-identifying Association Characteristic clause for non-key characteristics.

6. For each association, the same roles are defined in each state and are filled by the same entity.

This is satisfied in Theorem 10.1 by the Existence and Identity clause for restricted roles and the Non-identifying Association Role clause for unrestricted roles.

End of Proof.

10.2 Proof of Data Model Equivalence

The following theorem asserts the state-dependent equivalence of the semantic graph and the restricted semantic relation data models. Its proof is quite lengthy. However, an outline of the proof will be presented to show the essential ideas of the proof. The discussion following the proof should be informative even if the proof itself is not read in detail.

Theorem 10.2: The restricted semantic relation data model and the semantic graph data model are state-dependent equivalent.

Proof of Theorem 10.2:

As mentioned in the introduction to this chapter, this proof is based on an inductive proof presenting the constructions of equivalent operations to show the state-dependent equivalence of any pair of semantic graph and semantic relation application models where the relational schema was derived from the graph schema by Algorithms 8.1 and 9.1. We will show equivalence for any such pair of application models.

The application model equivalence relation so defined is onto the semantic graph data model because Algorithms 8.1 and 9.1 allow the definition of restricted semantic relation schemas for any graph schema. This application model equivalence relation is also onto the restricted semantic relation data model since every restricted semantic relation schema is derived from some graph schema by Algorithms 8.1 and 9.1. This will prove Theorem 10.2 according to the definition of state-dependent data model equivalence, Definition 7.7.

Consider semantic graph application model

$$g\text{-app} = (g\text{-schema}, g\text{-state}_0, g\text{-op-types})$$

where:

1. $g\text{-state}_0$ is the empty state.
2. $g\text{-op-types}$ defines the semantic graph operation types.

and semantic relation application model

$$r\text{-app} = (r\text{-schema}, r\text{-state}_0, r\text{-op-types})$$

where:

1. $r\text{-schema}$ is derivable from $g\text{-schema}$ by Algorithms 8.1 and 9.1
2. $r\text{-state}_0$ is the empty state.
3. $r\text{-op-types}$ defines the restricted semantic relation operation types.

Then $g\text{-app}$ and $r\text{-app}$ are shown to be state-dependent equivalent by the construction of state-dependent equivalent operations. An outline of the proof follows.

I. Show that initial states $g\text{-state}_0$ and $r\text{-state}_0$ are equivalent.

II. Show for any given pair of valid equivalent states $g\text{-state}$ and $r\text{-state}$, and a given valid semantic graph operation $g\text{-op}$, where $g\text{-op}(g\text{-state}) = g\text{-state}'$, that a valid relational operation $r\text{-op}$ can be constructed so that $r\text{-state}'$, where $r\text{-op}(r\text{-state}) = r\text{-state}'$, is equivalent to $g\text{-state}'$. Note that we will not need to use composed relational operations to obtain the necessary equivalence properties. Consider the following cases:

A. $g\text{-state}' \neq \text{error}$

1. The type of $g\text{-op}$ is Insert-ind-entity, Insert-ind-association, or Insert-semantic-unit.

2. The type of $g\text{-op}$ is Delete-ind-entity, Delete-ind-association, or Delete-semantic-unit.

B. $g\text{-state}' = \text{error}$

III. Show for any given pair of valid equivalent states $g\text{-state}$ and $r\text{-state}$, and a given valid semantic relation operation $r\text{-op}$, where $r\text{-op}(r\text{-state}) = r\text{-state}'$, that a composition of valid graph operations $g\text{-op}^*$ can be constructed so that $g\text{-state}'$, where $g\text{-op}^*(g\text{-state}) = g\text{-state}'$, is equivalent to $r\text{-state}'$. Consider the following cases:

A. $r\text{-state}' \neq \text{error}$

1. $r\text{-op}$ is of the type Insert-tuples.

2. $r\text{-op}$ is of the type Delete-tuples.

B. $r\text{-state}' = \text{error}$

The complete constructions follow along with justifications for their correctness.

Examples based on the operations of previous chapters will be interspersed.

I. The initial states $g\text{-state}_0$ and $r\text{-state}_0$ are empty states. These are clearly equivalent according to Theorem 10.1.

II. Show for any given pair of valid equivalent states $g\text{-state}$ and $r\text{-state}$, and a given valid semantic graph operation $g\text{-op}$, where $g\text{-op}(g\text{-state}) = g\text{-state}'$, that a valid relational operation $r\text{-op}$ can be constructed so that $r\text{-state}'$, where $r\text{-op}(r\text{-state}) = r\text{-state}'$, is equivalent to $g\text{-state}'$. Consider the following cases:

A. If $g\text{-state}' = \text{error}$, then consider the valid relational state $r\text{-state}'$ which is equivalent to $g\text{-state}'$. This is uniquely defined for the given $r\text{-schema}$.

The uniqueness of $r\text{-state}'$ is guaranteed for the following reasons. $g\text{-state}'$ defines a unique set of base relations according to Theorem 10.1. The set of base relations defines a unique database state $r\text{-state}'$ according to the constraints defined in step III of Algorithm 9.1 which express each database relation in terms of the base relations. The uniqueness of the database relations follows from the fact that in each instance of a Case-join or a Predicate-join, the columns which are matched for the join include the identifying characteristics of the entity or association of concern. Constraints from step II of Algorithm 9.1 guarantee that the base relations have equal sets of identifying characteristics.

1. If the type of g-op is Insert-ind-entity, Insert-ind-association or Insert-semantic-unit, then let r-op be

operation(*r-state*:R-state)
returns R-state

return Insert-tuples(*r-schema*, *r-state*, addition)

where addition = $\lambda r-n.(r\text{-state}'(r-n) - r\text{-state}(r-n))$. That is, the operation simply inserts those tuples in *r-state'* which do not exist in *r-state*. This operation applied to *r-state* will result in *r-state'*. *r-state'* is by construction equivalent to *g-state'*.

Note however, that there may be tuples in *r-state* which do not exist in *r-state'*. Those would be tuples with null values where added tuples take precedence. These tuples would automatically disappear, but the information in them would still be present in the tuples which replaced them.

For example, the graph state of Figure 5.2 is equivalent to the relational state of Figure 6.6. The graph state of Figure 5.6 is equivalent to the relational state of Figure 6.11. The relational operation 6-4 is then equivalent to the graph operation 5-2 since 6-4 inserts the single tuple which is in Figure 6.11 but not in Figure 6.6. This is a case in which the inserted tuple takes precedence over an existing tuple in the relation.

2. If the type of g-op is Delete-ind-entity, Delete-ind-association or Delete-semantic-unit, then let r-op be

```
operation(r-state:R-state)
  returns R-state
```

```
  return Delete-tuples(r-schema, r-state, deletion)
```

where deletion is defined by considering the projections of r-state' into the base relations of r-state'. Also consider the projections of r-state into the base relations of r-state.

a. Define r-state'₁ in the following manner. Consider the set of base relations which, for each base relation, consists of the set difference between the base relation of r-state and the corresponding base relation of r-state' (base relation of r-state - base relation of r-state'). The base relations will have the tuples describing each deleted entity or association. Add to these base relations those tuples in the types 1 and 4 base relations of r-state' describing each entity which was itself not deleted, but was in a deleted association. These tuples are identified by matching the characteristic in the type 1 relation for the predicate corresponding to the association. r-state'₁ is then uniquely defined by these base relations and the constraints which express each database relation in terms of the base relations. Note that r-state'₁ may not itself be a valid state.

b. Let $r\text{-state}'_2$ be $r\text{-state}'_1$ except for the deletion of all of those tuples which are less than or equal to any tuple in the corresponding relation in $r\text{-state}'$. That is, we want to specify a set of tuples consisting of all of those statements which are true of the objects deleted from the graph state and which are not true of entities or associations remaining in the graph state.

c. Finally, for each tuple in $r\text{-state}'_2$ which has the be e:object predicate:case pair present along with any other predicate:case pair in the same column group, if the characteristics in the tuple are equal to the characteristics in any of the tuples in the types 1 or 4 base relations which described entities which were not themselves deleted, but were in deleted associations, then eliminate be e:object from the column group's set of predicate:case pairs. Let the resulting state be deletion. This is done because while we may need information about these entities to describe the deleted association, we do not want to delete the information describing the non-deleted entities.

As an example, consider operation 5-5. We want to derive an equivalent relational operation. We have claimed that operation 6-14 is such an operation. These operations map equivalent states 6.11 and 5.6 into states 6.6 and 5.2. Figures 10.1 and 10.2 show the Supervise-operate relation before ($r\text{-state}$) and after ($r\text{-state}'$) the deletion.

The difference between the base relations of state 6.11 and state 6.6 is

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	NZ745	lathe
G.Wayshum	T.Manhart	32	NZ745	lathe
G.Wayshum	C.Gershag	40	JCL181	press
---	G.Wayshum	50	---	---

Figure 10.1 - Relation before Deletion

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
---	B.Belgim	35	NZ745	lathe
G.Wayshum	T.Manhart	32	NZ745	lathe
G.Wayshum	C.Gershag	40	JCL181	press
---	G.Wayshum	50	---	---

Figure 10.2 - Relation after Deletion

the single tuple shown in Figure 10.3 with empty sets for all of the other base relations. We then add the tuples describing the entities in the deleted association to get Figure 10.4 (with empty sets for those base relations not pictured). These base relations then correspond to the state shown in

Figure 10.5.

Supervise-base	
supervise:agent	supervise:object
employee	employee
name	name
names	names
G.Wayshum	B.Belgin

Figure 10.3 - Difference in Base Relations

Then we delete from Figure 10.5 ($r\text{-state}'_1$) the second tuple in Supervise-operate since it also occurs in Figure 10.2 ($r\text{-state}'$) to get Figure 10.6 ($r\text{-state}'_2$). Since in the one remaining tuple, the values in the column containing the be employee:object pair are equal to those in one of the tuples added to the base relations of Figure 10.3 (resulting in Figure 10.4), be employee:object is removed from the heading of the tuple to result in Figure 10.7 (deletion). This is the single tuple deleted in operation 6-14.

Supervise-base

supervise:agent	supervise:object
employee	employee
name	name
names	names
G.Wayshum	B.Belgim

Employee-char-base

be employee:object	
employee	
name	age
names	years
G.Wayshum	50
B.Belgim	35

Be-employee-base

be employee:object
employee
name
names
G.Wayshum
B.Belgim

Figure 10.4 - Addition to Base Relations

B. If $g\text{-op}(g\text{-state}) = \text{error}$, then let $r\text{-op}$ be

operation($r\text{-state}:\mathbf{R}\text{-state}$)
returns $\mathbf{R}\text{-state}$

return Insert-tuples($r\text{-schema}, r\text{-state}, \{(r\text{-n}, \{t\})\}$)

where $r\text{-n}$ is the name of any relation in $r\text{-schema}$ and t is a tuple mapping each characteristic defined in the relation into the null value.

This operation will map any $r\text{-state}$ into error. Hence $r\text{-op}$ and $g\text{-op}$ are equivalent.

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum ---	B.Belgim G.Wayshum	35 50	--- ---	--- ---

Figure 10.5 - Relation in r-state'₁

Supervise-operate

supervise:agent	be employee:object supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	---	---

Figure 10.6 - Relation in r-state'₂

Supervise-operate

supervise:agent	supervise:object operate:agent		operate:object	
employee	employee		machine	
name	name	age	number	type
names	names	years	serial-numbers	machine-types
G.Wayshum	B.Belgim	35	---	---

Figure 10.7 - Relation in deletion

III. Show for any given pair of valid equivalent states $g\text{-state}$ and $r\text{-state}$, and a given valid semantic relation operation $r\text{-op}$, where $r\text{-op}(r\text{-state}) = r\text{-state}'$, that a composition of valid graph operations $g\text{-op}^*$ can be constructed so that $g\text{-state}'$, where $g\text{-op}^*(g\text{-state}) = g\text{-state}'$, is equivalent to $r\text{-state}'$. Consider the following cases:

A. If $r\text{-state}' \neq \text{error}$, then consider the set of base relations corresponding to $r\text{-state}$ and the set of base relations corresponding to $r\text{-state}'$.

1. For the case that $r\text{-op}$ is of the type Insert-tuples:

a. If, among the set of type 1 base relations defining the restricted roles and key characteristics of the predicates, only a single tuple t with the predicate p has been added to the base relations of r -state to get the base relations of r -state', then:

i. If p is of the form $be\ e$ for some entity type e , then a single independent entity has been inserted. Let $g\text{-op}^*$ be

```
operation( $g$ -state: $g$ -state)
  returns  $g$ -state
```

```
  return Insert-ind-entity( $g$ -schema, $g$ -state, $e$ -desc)
```

where $e\text{-desc.E-id} = (e, t(((be\ e, object)), e))$
and $e\text{-desc.chars} = \bigcup_i t_i(((be\ e, object)), e)$

where the t_i are the i tuples in the types 1 and 4 base relations describing the characteristics of the added entity. By Theorem 10.1, we know that such tuples must also have been added. This results in an equivalent graph operation inserting the entity.

ii. If p is not of the form $be\ e$, then a single independent association has been inserted. Let $g\text{-op}^*$ be

```
operation( $g$ -state: $g$ -state)
  returns  $g$ -state
```

```
  return Insert-ind-association( $g$ -schema, $g$ -state, $a$ -desc)
```

where $a\text{-desc.A-id} = (Tp^{-1}(p), roles, t(Tp^{-1}(p)))$

where $roles \subset \text{Role-function}$

and $\text{dom}(roles) =$

$\{r:R \mid (\exists e:E) (((Tp^{-1}(p), Ts(r))), e) \in \text{dom}(t))\}$

and $(\forall r:R) (r \in \text{dom}(roles) \supset$

$(roles(r).chars =$

$t(((Tp^{-1}(p), Ts(r))), e))$

and $\text{dom}(a\text{-desc}.roles) = \text{dom}(roles) \cup$

$\{r:R \mid \text{type 3 base relations exist describing the unrestricted case } Ts(r) \text{ of the predicate } p\}$

and $(\forall r:R) (r \in \text{dom}(a\text{-desc}.roles)$

$\supset a\text{-desc}.roles(r) = t_i(((Tp^{-1}(p), Ts(r))), e))$

where the t_i are the type 1 tuple and the tuples added to the base relation for each unrestricted case. Such tuples must exist due to Definition 6.1.

and $\text{dom}(\text{a-desc.chars}) = \text{dom}(t(Tp^{-1}(p)))$

$\cup \{c:C \mid \text{type 2 base relations exist describing non-key characteristics of } p\}$

and $(\forall c:C) (c \in \text{dom}(\text{a-desc.chars})$

$\supset \text{a-desc.chars}(c) = t_i(Tp^{-1}(p))(c))$

where t_i is the tuple added to the base relation for each non-key characteristic. Such tuples must exist due to Theorem 10.1. This results in an equivalent graph operation inserting the association.

b. If several tuples have been inserted into type 1 base relations, then several entities and/or associations have been inserted.

i. These tuples can be partitioned according to the following rules:

1). Every tuple added to be a type 1 base relations for which the entity type e is a mandatory case of some predicate p (other than be e) must be in the same partition as the tuple in the type 1 base relation for p which has the identical entity type and characteristics for the mandatory case.

2). Any tuple which is not required to be in some partition by rule 1). above must be in a partition which contains the tuple as its only member.

This partitioning is necessary because a single relational operation may add information to the database state which equivalently requires several semantic graph operations. The partitioning divides the added type 1 tuples into sets which specify either added semantic units, added independent entities or added independent associations.

ii. The equivalent graph operation is then defined as the composition, in order, of network operations adding each independent entity, independent association or semantic unit.

Let

$$g\text{-op}^* = g\text{-op}_1 \circ g\text{-op}_2 \circ \dots \circ g\text{-op}_n$$

where each $g\text{-op}_i$ satisfies one of the following cases:

- 1). If the partition consists of a single tuple from a type 1 base relation for a be e predicate, then an independent entity has been added. Let $g\text{-op}_i$ be defined for this tuple as in 1.a.i with the provision that tuples which define non-key characteristics must have key characteristics identical to the tuple in the partition.

2). If the partition consists of a single tuple from a type 1 base relation for a predicate other than *be e*, then an independent association has been added. Let *g-op_i* be defined for this tuple as in 1.a.ii with the provision that restricted case key characteristics and predicate key characteristics in the tuples defining unrestricted case and non-key characteristics must match those of the tuple in the partition.

3). If the partition consists of more than one tuple, then a semantic unit has been added. Then let *g-op_i* be

```
operation(g-state:g-state)  
  returns g-state
```

```
  return Insert-semantic-unit(g-schema,g-state,arg-list)
```

where *arg-list* is defined as follows after first arbitrarily picking one of the *be e* tuples in the partition and beginning the procedure with an empty *arg-list*.

a). Generate an *e-desc* for the entity as specified in step 1.a.i and append this to the end of the *arg-list*.

b). For each association in which this entity type is mandatory (other than *be e*), define an *a-desc* for the predicate of this association type according to 1.a.ii. If this *a-desc* is not already in *arg-list*, then add it to *arg-list* with the requirement that firstly *e-desc*'s are

generated according to a. above for each entity tuple which is being added in the same partition and these e-desc's are added to *arg-list* if they are not already in the list. The e-desc's are added according to the ordering on role names specified by next-role. This step's generation of a-desc's is to be ordered by the ordering on association names specified by next-assoc-role.

As an example, consider the relational operation 6-1. We want to derive an equivalent graph operation. We have claimed that 5-3 is such an operation. The operations map the equivalent states of Figures 5.4 and 6.7 into the equivalent states of Figures 5.7 and 6.8.

The base relations corresponding to state 6.7 are shown in Figure 10.8. The base relations corresponding to state 6.8 are shown in Figure 10.9. The partitioning of the tuples added to the type 1 base relations results in a single partition, a tuple from Be-machine-base and a tuple from Operate-base, since operate:object is a mandatory case for the machine entity type. So a single semantic unit has been added and the procedure as specified above is to be followed.

An *arg-list* must be defined which first describes the added entity and then describes the added association. The straightforward result of this procedure for the base relations of Figures 10.8 and 10.9 is the *arg-list* used in operation 5-3.

Be-machine-base

be machine:object
machine
number
serial-numbers
NZ745 JCL181

Be-employee-base

be employee:object
employee
name
names
T.Manhart C.Gershag G.Wayshum

Operate-base

operate:object
machine
number
serial-numbers
NZ745 JCL181

Operate-case-base

operate:agent	operate:object
employee	machine
name	number
names	serial-numbers
T.Manhart C.Gershag	NZ745 JCL181

Employee-char-base

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Machine-char-base

be machine:object	
machine	
number	type
serial-numbers	machine-types
NZ745 JCL181	lathe press

Figure 10.8 - Base Relations before Insertion

Be-machine-base

be machine:object
machine
number
serial-numbers
NZ745 JCL181 AB982

Be-employee-base

be employee:object
employee
name
names
T.Manhart C.Gershag G.Wayshum

Operate-base

operate:object
machine
number
serial-numbers
NZ745 JCL181 AB982

Operate-case-base

operate:agent	operate:object
employee	machine
name	number
names	serial-numbers
T.Manhart C.Gershag G.Wayshum	NZ745 JCL181 AB982

Employee-char-base

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50

Machine-char-base

be machine:object	
machine	
number	type
serial-numbers	machine-types
NZ745	lathe
JCL181	press
AB982	lathe

Figure 10.9 - Base Relations after Insertion

2. For the case that $r\text{-op}$ is of the type Delete-tuples:

a. If, among the set of type 1 base relations defining the restricted roles and key characteristics of the predicates, only a single tuple t with the predicate p has been deleted from the base relations of $r\text{-state}$ to get the base relations of $r\text{-state}'$, then

i. If p is of the form $be\ e$ for some entity type e , then a single independent entity has been deleted. Let $g\text{-op}^*$ be

```
operation( $g\text{-state}$ : $g\text{-state}$ )
  returns  $g\text{-state}$ 

  return Delete-ind-entity( $g\text{-schema}$ ,  $g\text{-state}$ ,
    ( $e$ ,  $t(\{(be\ e, object)\}, e)\)$ )
```

This specifies the type and the key characteristics of the entity to be deleted.

ii. If p is not of the form $be\ e$, then a single independent association has been deleted. Let $g\text{-op}^*$

```
operation( $g\text{-state}$ : $g\text{-state}$ )
  returns  $g\text{-state}$ 

  return Delete-ind-association( $g\text{-schema}$ ,  $g\text{-state}$ ,  $a\text{-id}$ )
```

```
where  $a\text{-id} = (Tp^{-1}(p), roles, t(Tp^{-1}(p)))$ 
where  $roles \in \text{Role-function}$ 
and  $\text{dom}(roles) =$ 
   $\{r:R \mid (\exists e:E) (\{(Tp^{-1}(p), Ts(r))\}, e) \in \text{dom}(t)\}$ 
and  $(\forall r:R) (r \in \text{dom}(roles) \supset$ 
   $(roles(r).chars =$ 
     $t(\{(Tp^{-1}(p), Ts(r))\}, e)\)$ 
```

This identifies the association to be deleted.

b. If several tuples have been deleted from type 1 base relations, then several entities and/or associations have been deleted.

i. These tuples can be partitioned as specified in 1.b for the Insert-tuples case. Similar to that case, it means here that some combination of semantic units, independent entities and independent associations have been deleted.

ii. The equivalent graph operation is then defined as the composition, in order, of network operations deleting each semantic unit, independent association or independent entity.

Let

$$g-op^* = g-op_1 \circ g-op_2 \circ \dots \circ g-op_n$$

where each $g-op_i$ satisfies one of the following cases:

1). If the partition consists of a single tuple from a type 1 base relation for a be e predicate, then an independent entity has been deleted. Let $g-op_i$ be defined for this tuple as in 2.a.i.

2). If the partition consists of a single tuple from a type 1 base relation for a predicate other than be e, then an independent association has been deleted. Let $g-op_i$ be defined for this tuple as in 2.a.ii.

3). If the partition consists of more than one tuple, then a semantic unit has been deleted. Then let $g-op_1$ be

operation($g-state:g-state$)
returns $g-state$

return Delete-semantic-unit($g-schema, g-state, a-id$)

where $a-id$ is defined as in 2.a.ii for any arbitrarily chosen tuple in the partition representing an association in the partition (i.e. a predicate other than $be\ e$). This is sufficient to define a semantic graph operation which will delete the entire semantic unit.

B. If $r-state' = error$, then let $g-op^*$ be

operation($g-state:g-state$)
returns $g-state$

return Insert-ind-entity($g-schema, g-state, e-desc$)

where $e-desc \in E-desc$ has a $chars$ component which maps some characteristic into a domain element (d, str) such that $g-schema.sd(d)(str) = false$. That is, the value is not a valid element of the domain.

This operation will map any $g-state$ into error. Hence $g-op^*$ and $r-op$ are equivalent.

This completes the inductive proof of application model equivalence. Since it is applicable to any pair of application models consisting of a semantic graph application model and a restricted semantic relation application model whose schema was derived from the graph model, data model equivalence has been shown between the semantic graph data model and the restricted semantic relation data model.

This ends the proof of Theorem 10.2.

10.3 Discussion

At this point we will summarize the basic ideas behind the constructions presented in the proof of Theorem 10.2. Part I simply states the equivalence of the empty states. In part II a construction is presented to find a relational operation equivalent to a given graph operation. We know which relational state is equivalent to the graph state resulting from the graph operation, so we work backwards from this to find the necessary relational operation.

In the case in which entities or associations were added to the graph state, an Insert-tuples operation is constructed which simply adds those tuples present in the desired relational state but not present in the initial relational state. In the case that entities or associations were deleted from the graph state, a Delete-tuples operation is constructed which deletes all of those statements true in the initial relational state but not true in the desired state. Both the Insert-tuples and Delete-tuples operations make use of the partial ordering defined on tuples.

In part III a construction is presented to find a graph operation equivalent to a given relational operation. The relational operation is found by examining the base relations of the two relational states. From the base relations, descriptions are derived of the entities and/or associations which have been inserted or deleted. A composed operation is necessary if more than one independent entity, independent association or semantic unit is to be inserted or deleted.

Note that the construction in part III could have been done as in part II by first finding the graph state which is the desired result of the graph operations and then working backwards from there to find the necessary graph operation. The construction presented was chosen because it better illustrated how the information about the state is distributed throughout the base relations.

We will now make several observations concerning these constructions. Firstly, recall that state-dependent equivalence allows the use of composed operations. For the two data models studied here, we needed to construct composed operations only for the semantic graph model since a single relational operation can insert an arbitrary number of entities and associations.

Secondly, we can see that the state-dependence required for the constructions arises from the nature of the semantic relation data model. The proper form of an insertion or deletion is dependent upon the database state being modified.

Finally, it should be noted that the handling of operations which result in the error state is not completely satisfactory. In particular, all errors are

treated identically. An actual implementation would certainly distinguish different kinds of errors.

Chapter 11 - Conclusions

11.1 Summary

The motivation for this research was to gain an understanding of data model equivalence properties. Such an understanding is necessary if one wishes to implement any of:

1. An ANSI-like database interface providing, for a single database, multiple views based on different data models.
2. A common interface to several database systems which are defined in terms of different data models.
3. The transference of both databases and programs between systems based on different data models.

We were also also interested in gaining insight into:

4. The evaluation of data model expressiveness.
5. The relational vs. network data model "controversy".

To convincingly attack these issues, the notion of a *semantic data model* was introduced. Specifically, the *semantic graph* and *semantic relation data models* were defined in Chapters 2, 5 and 6. These data models are meant to be semantic versions of the network and relational data models. The structures used in semantic data models are meant to have clear interpretations in terms of

the applications being modelled. The semantics of the two data models defined in this thesis are expressed by natural language *case grammars*.

Each object in a semantic graph database state is meant to represent an object in the state of the application being modelled. The operation types allowed are the insertion and deletion of *entities*, *associations* or *semantic units*.

A semantic relation database state consists of *statements* (tuples) describing the state of the application. The allowed operation types are the insertion and deletion of sets of such statements. The insertion and deletion operations are defined in terms of a *partial ordering* of statements.

These two data models were defined in terms of a *formal framework for data model definition* as presented in Chapter 4. This framework is based on the notions of *application model*, *schema*, *operation type*, *database state* and *database*. The framework is applicable to syntactic data models as well as semantic data models. However, the task of comparing data models is claimed to be easier for semantic data models.

Chapter 7 specifies the various equivalence properties of interest. *Data model equivalence* is based on the notions of *application model equivalence*, *operation equivalence* and *database state equivalence*. It was shown that *composed operation* and *state dependent equivalence* must be considered in addition to *isomorphic equivalence*. The choice between isomorphic, composed operation and state dependent equivalence for a specific multiple data model system impacts the types of implementations which can be used.

Due to the generality of the semantic relation data model constraints, the

semantic graph and semantic relation data models are not totally (state dependent) equivalent. A data model, the *restricted semantic relation data model*, which is totally equivalent to the semantic graph data model was then defined.

Chapter 8 showed how to define all but the constraint component of restricted semantic relation application models. The essence of this procedure was the definition of a semantic relation as containing statements describing a particular portion of a semantic graph application. It is possible to define several relational application models for a single graph application model. Such a feature seems appropriate for database systems providing differing views of the same application. It is also possible that a semantic relation application model is redundant, containing the same information in several relations. The user may desire to combine statements concerning the application in many different ways.

The restricted semantic relation constraint definition algorithm was presented in Chapter 9. Five types of constraints are used: Equal, Equalc, Subset, Non-null and Col-func. Constraints are stated in terms of a semantic relational algebra. The operations in this algebra are Case-join, Predicate-join, Conjunction and Project. It was observed that a large number of relational constraints are required for even a small application.

The state dependent equivalence of the semantic graph and semantic relation data models was proved in Chapter 10. Equivalence was shown for each pair of application models consisting of a semantic graph application model and a semantic relation application model whose schema was derived from the

graph schema using the algorithms presented in Chapters 8 and 9. Equivalence for each such pair of application models implied equivalence of the two data models. Application model equivalence was shown by presenting algorithms to construct equivalent operations for the application models.

11.2 Relational vs. Network: Some Observations

The various arguments in favor of either the relational or network syntactic data models have counterparts in terms of the semantic data models. We feel that our examination of the equivalence problems of the semantic data models gives some insight for considering the validity of these arguments. We will comment on this "controversy".

Firstly, we can state that when restricting the data models to equivalent graph and relational application models, the "expressive powers" of the models are equivalent. Each state and each operation in one model has an equivalent counterpart in the other data model. So comments on whether the graph or relational view is "better" are strictly subjective unless the comparison is dependent upon the existence of relational application models with no equivalent graph model. We will discuss this facet of the comparison later.

We can look at two major facets of the capabilities provided by a database system - retrieval and update. Recall from Chapter 3 that one of the main distinctions drawn in the literature between network and syntactic relational database systems is in terms of set-oriented or "one-at-a-time" retrieval

languages. As pointed out in Chapter 3, this is really an argument concerning efficiency of implementation and either type of language could be used with either type of data model. The same points hold for the semantic data models.

Proponents of the syntactic relation model claim that the relational model is much easier for the user to comprehend since it presents only a single type of structure, relations, while the network models present several different types of structures [Date and Codd]. We can see from examining the formal state specifications of the semantic graph and semantic relation data models that the same "simplicity" of the relational view is reflected in its less complex specification.

Advocates of the syntactic relational model also claim that update operations are also quite simple and clear because there are only two simple types of operations - adding or deleting tuples [Date and Codd]. We can see in our semantic models that the formal definitions of the relational operation types are certainly less complex than the definitions of the graph operation types. But this argument completely ignores the complexity of the large number of constraints necessary to maintain the semantic integrity in a relational database. An attempt at performing updates to a semantic relation database without a knowledge of the constraints could be a difficult process. In the graph model, knowing what type of operation to use recognizes that certain types of predefined requirements must be satisfied by the update operation. One way or another the complexity of the application must be represented in the application model. The choice of which data model to use can significantly alter where

the complexity is to reside.

11.3 *Foundation for a User Interface*

As just stated, one way or another the complexity of the application must be represented in the application model; the choice of which data model to use can significantly alter where the complexity is to reside. While the semantic relation data model provides a nice tabular view of the database state, it seems poor for performing updates. The semantic graph data model allows an easier understanding of updates, but would be quite unwieldy for retrievals other than those involving a small amount of data. However, the ability to support equivalent relational and graph application models accessing a shared database would allow the best of both worlds - a simple relational view for retrieval and a graph model for updating.

The data model equivalence properties proved in this thesis for the semantic graph and restricted semantic relation data models guarantee that this ability could be provided in a single system. The system could also accommodate those users desiring retrievals in terms of the semantic graph model and updates in terms of relations. We propose here the construction of a database system which provides as a user interface a *dual semantic data model* consisting of the semantic graph and restricted semantic relation data models.

For the user of this system, database design would begin with the construction of a semantic graph schema. Based on this schema, a procedure

similar in principle to Algorithm 8.1 would be followed to "encircle" portions of the graph schema. Each such portion would define a relation.

We saw earlier in this thesis that a natural language sentence template can be associated with each relation. We envision that these templates would be automatically generated for each relation. This automatic generation is much like the generation of sentences from "deep structure" representations in artificial intelligence natural language systems [Bruce]. The generation of the template would serve as feedback during the design process. The designer would examine the templates to confirm that the relations contain the desired information in the proper form. After the design stage, the templates would be displayed with the relations. The templates do not replace the relation headings. Rather, they show the same information in a different form.

Since there are several relational application models equivalent to each graph model, we could then extend such a system to let different users see different relational versions of the same application. A user can retrieve from and operate on the database state in terms of either semantic graphs or semantic relations.

While the specification of retrieval languages has not been essential to this thesis, the design of such languages would be necessary for the proposed system. [Deheneffe and Hennebert] present a retrieval and update language for a semantic graph data model similar to the one presented in this thesis. We are not aware of any retrieval language appropriate for a semantic relation data model as presented in this thesis. Just as the relational algebra is a base for a

retrieval language for the syntactic relational model [Codd 72b], the operations used in constraint definition in this thesis might be considered an appropriate starting point for an algebraic language.

[Chen] provides brief examples of a retrieval language for his data model similar to the mapping language of [Chamberlin and Boyce] used for syntactic relational systems. The sentence templates would be particularly appropriate for retrieval in a manner similar to Query-by-Example [Zloof]. For updates, a fill-in-the-blank approach using the sentence templates seems natural. Any help provided through research in the use of natural language as a database interface would obviously be beneficial.

Semantic relations are assumed to always contain *all* possible statements of the form specified for the relation. If we allow retrievals to dynamically create new relations, we must be sure to properly modify the relation heading and template. For example, consider the relation describing employees in Figure 11.1. Now assume we want to retrieve the statements from this relation describing those employees whose age is less than 40 years. To simply present to the user the relation in Figure 11.2 would be misleading unless a clear distinction was made between database relations and retrieved relations. The user should be aware of the fact that a particular relation contains only a subset of the possible statements fitting its template.

We propose investigating the possibility of reflecting the semantics of the retrieval in the relation heading as shown in Figure 11.3 for this example. Figure 11.4 shows the result of a projection of the employee names in

"There is an employee with name ____ and age ____."

be employee:object	
employee	
name	age
names	years
T.Manhart	32
C.Gershag	40
G.Wayshum	50
B.Belgin	35

Figure 11.1 - Semantic Relation

"There is an employee with name ____ and age ____."

be employee:object	
employee	
name	age
names	years
T.Manhart	32
B.Belgin	35

Figure 11.2 - Result of Retrieval

Figure 11.3. The result of a projection of the employee ages in Figure 11.3 results in Figure 11.5. Note the sentence templates in these figures. Exploring the details of this approach is an area for future research.

"There is an employee aged less than 40 with name ____ and age ____."

be employee aged less than 40:object	
employee	
name	age
names	years
T.Manhart	32
B.Belgim	35

Figure 11.3 - Retrieval Result Showing Semantics of Retrieval

We have presented here only an incomplete description of the dual semantic data model interface. The directions outlined seem natural extensions of the research in this thesis.

The ability to define the dual data model also suggests several other capabilities which can be provided in a database system. We could present a strictly relational model to one user and a strictly graph model to a different user. The database system can maintain one of the components of the dual

"There is an employee aged less than 40 with name ____."

be employee aged less than 40:object
employee
name
names
T.Manhart
B.Belgim

Figure 11.4 - Projection of Figure 11.3

"There is an employee aged less than 40 with age ____."

be employee aged less than 40:object
employee
age
years
32
35

Figure 11.5 - Projection of Figure 11.3

database state hidden from any particular user.

11.4 Inferential Knowledge Representation Systems

We ask here whether the data model equivalence framework presented in this thesis is useful for the study of "knowledge representation language" databases used with inferential artificial intelligence systems. For example, one might wonder if a particular OWL Linguistic Memory System database [Szolovits et. al.] is equivalent to a particular FRL (Frame Representation Language) database [Roberts and Goldstein]. The answer to the question of usefulness is a qualified yes.

If an adequate definition of database state equivalence can be given for the data models of concern, then the rest of the framework can be used. However, the definition of database state equivalence for two such knowledge representation languages seems to have several obstacles.

Firstly, since these data models are designed to be used with inferential systems, there may be structures in the database state which are oriented towards a specific inference mechanism. For example, in OWL Linguistic Memory there are "multi-purpose" *links* whose meaning is dependent upon the workings of the OWL interpreter which uses the database. The necessity of considering the inference mechanisms for determining the meaning of the database is not encompassed by the framework used in this thesis.

Secondly, a great "richness" of representation is desired for such systems. In particular, it is desired to represent procedural knowledge. The mechanism used for such procedural representation may make it impossible to give clear

interpretations of such procedures at a semantic level. This is likely to be the case in FRL which allows the insertion of arbitrary LISP procedures into the database.

Finally, to define equivalence between data models there must be at some level a common view of the world. In this thesis, both the semantic graph and semantic relation data models are based on a simple entity-association view of the world. It is an open question whether enough commonality exists between more complex knowledge representation languages. [Smith] includes an exploration of the issues involved with this question.

11.5 Other Applications and Areas for Future Research

1. A formal framework for defining data models has been presented. Using this framework, several different types of data model equivalence were defined. This framework and these equivalence definitions should be used to evaluate the suitability of different data models for the different ANSI schema levels. Insight can be gained into both the data models themselves and into the type of implementation needed.

The considerations involved in actually implementing an ANSI-like architecture must include an examination of the equivalence issues discussed in this thesis. Whether the defined equivalences are state dependent or not would have a large effect on the types of implementations allowable. The operation equivalence mappings from the external to internal levels should be analyzed as

a guide for implementation efficiency considerations. The same types of equivalence mappings must be involved in the transportation of a database and associated programs from one database system to another.

There should be more development of semantic data models such as the semantic relation model for the external schema level. While it may be necessary to support existing syntactic data models at the external level, all users should not be deprived of important semantic information which helps them better understand how the application is modelled.

2. The complement of the previous suggestion for research is an examination of the issues involved in providing a common interface to differing database systems defined in terms of different data models. This problem is most applicable in attempting to interface existing independent systems into a *distributed system* or *network*. A particularly important (and perhaps vexing) feature of this problem is that most existing database systems are based on syntactic data models. Hence, the problem of defining state equivalence is difficult.

One interesting approach to this problem would be using the dual semantic data model interface previously described as the common interface. The database states of existing network, hierarchical and relational systems would be viewed in terms of semantic graphs and relations. To do this would quite likely require the procurement of semantic information concerning the existing applications.

3. We saw in Chapter 8 that there are relational application models which have no equivalent graph application models, yet which seemed to specify potentially desirable applications. It seems worthwhile to explore how to define an extendible semantic graph model in which new types of semantic objects and appropriate operations can be specified. How this would relate to the derivation of equivalent constraints would need to be explored. The concept of abstract data types [Liskov et. al.] seems to be a good framework for such an extendible model. [Weber 76] and [Weber 78] show work relating abstract data types to database problems.

An alternative to defining new operations would be to put constraints, similar to those of the relational models, as additional requirements to be satisfied by semantic graph operations. [Chen] proposes this, but we feel that this is not in keeping with the nature of the semantic graph model in which allowable types and operations should directly model the application.

4. We have avoided implementation questions, but they must be answered before any data model is actually used. The semantic graph and semantic relation data models do not seem to introduce any particular problems different from those of previously proposed data models.

However, the concept of basing a database system on a dual data model adds another level of complexity. The various trade-offs involved should be identified and explored. Note that we do not need to know whether the dual data model is "really" implemented in terms of graphs or relations. Such

concepts are absent from data independent data models. One implementation methodology which claims to be good for both network and relational syntactic data bases is [Tsichritzis].

There are also the questions of concurrent access to a shared database and distributed database systems. We would want to guarantee that each user always sees a database system acting consistent with the definition of the data model on which it is based. [Hawryszkiewicz] uses a formal model of computation to prove the correctness of a theoretical implementation of a syntactic relational database system without constraints. This topic also needs to be explored in terms of semantic data models as presented in this thesis. The interaction of the semantics of the database state and the choice of "locking" units for concurrent users should be explored. The significance of database semantics to the implementation of distributed database systems is completely unexplored.

5. We can define the equivalence of data models only after we understand the semantics of the applications representable in the data model. For the same reason, efforts aimed at finding general methods of transferring data from one database system to another should be based on semantic data models. Work in the area of data translation (such as [Sibley and Taylor]) has been based on syntactic data models and has largely ignored the problems of transferring programs from one database system to another which requires the derivation of equivalent operations [Dennis].

This thesis has shown how equivalent database states and operations can be found for the two data models presented. The impact of equivalent semantic data models on the problem of data translation should be explored.

6. The examples used in this thesis have been relatively simple. In particular, note that only two cases (roles) have been used in expressing associations: agent and object. The expression of more complex associations might involve other cases [Bruce] such as result, instrument, source and goal. Attempts should be made to express other applications, preferably "real" ones, in terms of the semantic graph and semantic relation data models.

7. The entity-association model used as a semantic basis in this thesis is relatively simple. We will mention two specific areas in which extensions might be desirable.

Firstly, this model includes no specific concept of time. Time could be expressed as entity or association characteristics, but a more sophisticated approach might be useful. [Bradly] and [Bubenko] present data models incorporating time. There has also been pertinent work in the artificial intelligence area such as [Kahn] and [Thompson and Thompson].

Secondly, the models used in this thesis assume that entity sets are disjoint. Data models which do not make this assumption are described in [Palmer] and [Smith and Smith]. Again, researchers in artificial intelligence knowledge representation such as [Szolovits et. al.] and [Fahlman] have explored this topic.

11.6 Conclusion

There has been a twofold theme throughout this thesis: the importance of representing application semantics in a database system's data model and the necessity of being able to derive equivalent operations in two data models which purport to model the same class of applications. The themes are intertwined because only when it is agreed that the application semantics represented in two different data models are comparable can there be any meaningful definitions of equivalence.

We have presented in this thesis a formal framework for defining data models and data model equivalence. Two semantic data models have been presented and their various equivalences within the framework were shown. These included the concepts of state equivalence, operation equivalence, application model equivalence and data model equivalence. These various types of equivalences are not mere artifacts of formality, but rather are important properties which must be investigated whenever asserting that two data models are equivalent. This thesis was intended to convey the idea that a formal approach is necessary to truly understand the problems involved.

The two data models have merit in that they show how application semantics can be embedded in equivalent data models which are related in form to two widely discussed types of (syntactic) data models: network and relational. These data models are appropriate for database systems which propose to allow different users to have views of the database most appropriate for their

application. Moreover, it is shown how equivalent application models can be defined and how equivalent operations can be found for these equivalent application models.

It is not claimed that the data models described in this thesis are the "be all and end all" of data models. But it is hoped that this thesis will make the database system designer consider the issues of application semantics, data model equivalence, and their interrelations.

Bibliography

1. [ANSI] ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, in FDT, Vol. 7, No. 2, 1975.
2. [Astrahan et. al.] Astrahan, M.M., et. al., System R: Relational Approach to Database Management, ACM TODS, Vol. 1, No. 2, June 1976.
3. [Bachman 69] Bachman, C.W., Data Structure Diagrams, Data Base 1, 2, ACM SIGBDP, 1969.
4. [Bachman 73] Bachman, C.W., The Programmer as Navigator, CACM, Vol. 16, No. 11, Nov. 1973.
5. [Bachman and Daya] Bachman, C.W. and Daya, M., The Role Concept in Data Models, Proc. Int. Conf. on Very Large Data Bases, ACM, 1977.
6. [Bernstein] Bernstein, P.A., Normalization and Functional Dependencies in the Relational Data Base Model, Tech. Rep. CSRG-60, Comp. Systems Res. Group, U. of Toronto, Nov. 1975.
7. [Borkin] Borkin, S.A., Data Model Equivalence, Proc. Int. Conf. on Very Large Data Bases, IEEE, 1978.
8. [Bradly] Bradly, J., Operations Data Bases, Proc. Int. Conf. on Very Large Data Bases, IEEE, 1978.
9. [Bruce] Bruce, B., Case Systems for Natural Language, Artificial Intelligence 6, 1975.
10. [Bubenko], Bubenko, J.A., The Temporal Dimension in Information Modelling, Proc. IFIP TC-2 Working Conference, Modeling in Data Base Management Systems, IRIA, Nice, France, Jan. 1977.
11. [Bubenko et. al.] Bubenko, J.A. Jr., Berild, S., Lindencrona-Ohlin, E. and Nachmens, S., From Information Structures to DBTG Data Structures, Proc. Conference on Data: Abstraction, Definition and Structure, in ACM SIGPLAN Notices, Vol II, 1976.
12. [Chamberlin and Boyce] Chamberlin, D.D. and Boyce, R.F., SEQUEL: A Structured English Query Language, Proc. 1974 ACM SIGFIDET Workshop.

13. [Chamberlin et. al.] Chamberlin, D.D., Gray, J.N. and Traiger, I.L., Views, Authorization, and Locking in a Relational Data Base System, Proc. AFIPS 1975 NCC, Vol. 44.
14. [Chen] Chen, P.P., The Entity-Relationship Model - Toward a Unified View of Data, ACM TODS, Vol. 1, No. 1, March 1976.
15. [Codasyl] Codasyl Data Description Language Committee, Report, Information Systems, Vol. 3, No. 4, 1978.
16. [Codd 70] Codd, E.F., A Relational Model of Data for Large Shared Data Banks, CACM, Vol. 13, No. 6, June 1970.
17. [Codd 71] Codd, E.F., A Data Base Sublanguage Founded on the Relational Calculus, Proc. 1971 ACM SIGFIDET Workshop.
18. [Codd 72a] Codd, E.F., Further Normalization of the Data Base Relational Model, Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice Hall, 1972.
19. [Codd 72b] Codd, E.F., Relational Completeness of Data Base Sublanguages, Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice Hall, 1972.
20. [Codd 74] Codd, E.F., Recent Investigations in Relational Data Base Systems, Proc. IFIP Congress 1974.
21. [Date and Codd] Date, C.J. and Codd, E.F., The Relational and Network Approaches: Comparison of the Application Programming Interfaces, Proc. 1974 ACM SIGMOD Workshop.
22. [Date and Hopewell] Date, C.J. and Hopewell, P., Storage Structure and Physical Data Independence, Proc. 1971 ACM SIGFIDET Workshop.
23. [DBTG] Codasyl, Data Base Task Group Programming Language Committee, Proposal, ACM, Feb. 1973.
24. [Deheneffe and Hennebert] Deheneffe, C. and Hennebert, H., NUL: A Navigational User's Language For A Network Structured Data Base, Proc. ACM SIGMOD 1976.
25. [Deheneffe et. al.] Deheneffe, C., Hennebert, H. and Paulus, W., Relational Model for a Data Base, Proc. IFIP Congress 1974.

26. [Dennis] Dennis, J.B., On the Exchange of Information, Proc. ACM SIGFIDET 1970.
27. [Dennis et. al.] Dennis, J.B, Stoy, J.E. and Moss, E., paper in preparation.
28. [Engles] Engles, R.W., An Analysis of the April 1971 Data Base Task Group Report, Proc. 1971 ACM SIGFIDET Workshop.
29. [Fahlman] Fahlman, S.E., Thesis Progress Report: A System For Representing and Using Real-World Knowledge, Memo 331, M.I.T. Artificial Intelligence Laboratory, Cambridge, May 1975.
30. [Fehder] Fehder, P., HQL: A Set-Oriented Transaction Language for Hierarchically-Structured Data Bases, Proc. ACM 1974.
31. [Fillmore] Fillmore, C., The case for case, in Universals in Linguistic Theory, Bach and Harms, eds., Holt, Rinehart and Winston, 1968.
32. [Fleck] Fleck, M., On the Equivalence of Data Base Models, Technical Report TR 25.144, IBM Laboratory Vienna, July 1975.
33. [Gray et. al.] Gray, J.N., Lorie, R.A., Putzolu, G.R. and Traiger, I.L., Granularity of Locks and Degrees of Consistency in a Shared Data Base, Technical Report RJ.1654, IBM Research Laboratory, San Jose, September 1975.
34. [Hall et. al.] Hall, P., Owlett, J. and Todd, S., Relations and Entities, in Modelling in Data Base Management Systems, Nijssen, G.M., ed., North-Holland Publishing Company, New York, 1976.
35. [Hammer and McLeod] Hammer, M.M. and McLeod, D.J., Semantic Integrity in a Relational Data Base System, Proc. Int. Conf. on Very Large Data Bases, ACM, Sept. 1975.
36. [Hawryszkiewicz] Hawryszkiewicz, I.T., Semantics of Data Base Systems, TR-112, M.I.T. Laboratory for Computer Science, Cambridge, Dec. 1973.
37. [Kahn] Kahn, K.K., Mechanization of Temporal Knowledge, TR-155, M.I.T. Laboratory for Computer Science, Cambridge, Sept. 1975.
38. [Kay] Kay, M.H., An assessment of the Codasyl DDL for use with a relational subschema, in Data Base Description, Douque, B.C.M. and Nijssen, G.M., eds., North-Holland Publishing Company, New York, 1975.

39. [Kerschberg et. al.] Kerschberg, L., Klug, A. and Tsichritzis, D., A Taxonomy of Data Models, in Systems for Large Data Bases, Lockemann, P.C. and Neuhold, E.J., eds., North-Holland Publishing Company, New York, 1977.
40. [Klug and Tsichritzis] Klug, A. and Tsichritzis, D., Multiple view support within the ANSI/SPARC framework, Proc. Int. Conf. on Very Large Data Bases, ACM, 1977.
41. [Lamport] Lamport, L., Towards a Theory of Correctness for Multi-user Data Base Systems, Report CA-7610-0712, Massachusetts Computer Associates, Inc., Wakefield, Ma., Oct. 1976.
42. [Liskov et. al.] Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., Abstraction Mechanisms in CLU, CACM, Vol. 20, No. 8, August 1977.
43. [Martin] Martin, W.A., Descriptions and the Specializations of Concepts, TM-101, M.I.T. Laboratory for Computer Science, Cambridge, March 1978.
44. [McCarthy] McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, part 1, CACM, Vol. 3, No. 4, April 1960.
45. [McGee 74] McGee, W.C., A Contribution to the Study of Data Equivalence, in Data Base Management, Klimbie, J.W. and Koffeman, K.L., eds., North-Holland Publishing Company, New York, 1974.
46. [McGee 75] McGee, W.C., File-Level Operations on Network Data Structures, Proc. ACM SIGMOD 1975.
47. [McGee 77] McGee, W.C., The information management system IMS/VS, IBM Systems Journal 16, No. 2, 1977.
48. [Mylopoulos et. al.] Mylopoulos, J. et. al., TORUS - A Natural Language Understanding System for Data Management, Proc. IJCAI 1975.
49. [Navathe and Schkolnick] Navathe, S.B. and Schkolnick, M., View Representation in Logical Data Base Design, Proc. ACM SIGMOD 1978.
50. [Neuhold] Neuhold, E.J., Data Mapping: A Formal Hierarchical and Relational View, Bericht 10, Universitat Karlsruhe, Institut Fur Angewandte Informatik, Feb. 1973.
51. [Nijssen 74] Nijssen, G.M., Data Structuring in DDL and the Relational Data Model, in Data Base Management, Klimbie, J.W. and Koffeman, K.L., eds., North-Holland Publishing Company, New York, 1974.

52. [Nijssen 76] Nijssen, G.M. ed., Modelling in Data Base Management Systems, North-Holland Publishing Company, New York, 1976.
53. [Palmer] Palmer, I.R., Record Subtype Facilities in Database Systems, Proc. Int. Conf. on Very Large Data Bases, IEEE, 1978.
54. [Paolini and Pelagatti] Paolini, P. and Pelagatti, G., Formal Definitions of Mappings in a Data Base Proc. ACM SIGMOD 1977.
55. [Quillian] Quillian, M.R., Semantic Memory, in Semantic Information Processing, Minsky, M., ed., MIT Press, Cambridge, 1968.
56. [Sagiv and Yannakakis] Sagiv, Y. and Yannakakis, M., Equivalence Among Relational Expressions with the Union and Difference Operations, Proc. Int. Conf. on Very Large Data Bases, IEEE, 1978.
57. [Schank] Schank, R.C. and Colby, K.M., eds., Computer Models of Thought and Language, W.H. Freeman and Co., San Francisco, 1973.
58. [Schmid and Bernstein] Schmid, H.A. and Bernstein, P.A., A Multi-level Architecture for Relational Data Base Systems, Proc. Int. Conf. on Very Large Data Bases, ACM, Sept. 1975.
59. [Schmid and Swenson] Schmid, H.A. and Swenson, J.R., On the Semantics of the Relational Data Model, Proc. ACM SIGMOD 1975.
60. [Senko] Senko, M.E., Data Structures and Data Accessing in Data Base Systems Past, Present, Future, IBM Systems Journal 16, No. 3, 1977.
61. [Sibley] Sibley, E.H., On the Equivalence of Data Based Systems, Proc. ACM SIGFIDET 1974.
62. [Sibley and Taylor] Sibley, E.H. and Taylor, R.W., A Data Definition and Mapping Language, CACM, Vol. 16, No. 12, Dec. 1973.
63. [Simmons] Simmons, R.F., Semantic Networks: Their Computation and Uses for Understanding English Sentences, in Computer Models of Thought and Language, Schank, R.C. and Colby, K.M., eds., W.H. Freeman and Co., San Francisco, 1973.
64. [Smith] Smith, B.C., Levels, Layers, and Planes: A Framework for a Theory of Knowledge Representation Semantics, S.M. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Feb. 1977.

65. [Smith and Smith] Smith, J.M. and Smith, D.C.P., Database Abstractions: Aggregation and Generalization, ACM TODS, Vol. 2, No. 2, June 1977.
66. [Stonebraker] Stonebraker, M., High Level Integrity Assurance in Relational Data Base Management Systems, Memorandum ERL-M473, Electronics Research Laboratory, Univ. of Cal., Berkeley, August 1974.
67. [Stonebraker et. al.] Stonebraker, M. et. al., The Design and Implementation of INGRES, ACM TODS, Vol. 1, No. 3, Sept. 1976.
68. [Szolovits et. al.] Szolovits, P., Hawkinson, L.B. and Martin, W.A., An Overview of OWL, A Language for Knowledge Representation, TM-86, M.I.T. Laboratory for Computer Science, Cambridge, June 1977.
69. [Thompson and Thompson] Thompson, F.B. and Thompson, B.H., Practical Natural Language Processing: The REL System as Prototype, in Advances in Computers, Vol. 13, Rubinoff, M. and Yovits, M.C., eds., Academic Press, New York, 1975.
70. [Tsichritzis] Tsichritzis, D.C., LSL: A Link and Selector Language, Proc. ACM SIGMOD 1976.
71. [Weber 76] Weber, H., The D-graph model of large shared data bases: a representation of integrity constraints and views as abstract data types, Technical Report RJ1875, IBM Research Laboratory, San Jose, Nov. 1976.
72. [Weber 78] Weber, H., A Software Engineering View of Data Base Systems, Proc. Int. Conf. on Very Large Data Bases, IEEE, 1978.
73. [Wirth] Wirth, N., The programming language PASCAL, Acta Informatica, 1, 1971.
74. [Zimmerman] Zimmerman, K., Different Views of a Data Base: Coexistence Between Network Model and Relational Model, Technical Report TR 25.143, IBM Laboratory Vienna, June 1975.
75. [Zloof] Zloof, M.M., Query by Example, Proc. National Computer Conference, AFIPS, 1975.

Appendix 1. The Semantic Network Data Model: Formal Specifications

G-schema = (E,
A,
D,
r:((A × R) → E),
c:(((EUA) × C) → D),
fr:((A × R) → (restrict:{restricted,unrestricted},
mand:{mandatory,non-mandatory})),
fc:(((EUA) × C) → {key,non-key}),
sd:D → [Str → Boolean])
R,
C,
Str)

G-state = (E:Poweraset(E-id),
A:Poweraset(A-id),
D:Poweraset(Domain-element),
r:((A-id × R) → E-id),
c:((Id × C) → Domain-element))

Domain-element = (type:D, value:Str)
Char-function = [C → Domain-element]
E-id = (type:E, chars:Char-function)
Role-function = [R → E-id]
A-id = (type:A, roles:Role-function, chars:Char-function)
Id = A-id ∪ E-id
Boolean = {true, false}
Arg-list = (E-desc ∪ A-desc)* such that each element of the list is unique
E-desc = (E-id:E-id, chars:Char-function)
A-desc = (A-id:A-id, roles:Role-function, chars:Char-function)
E = entity set names
A = association set names
D = domain names
R = roles
C = characteristics
Str = character strings

Figure A1.1 - Type Specifications

first-role: $((A \times G\text{-schema}) \rightarrow R)$
next-role: $((A \times G\text{-schema} \times R) \rightarrow R)$
last-role: $((A \times G\text{-schema}) \rightarrow R)$

These three functions specify the ordering on the role set edges for a specific association set node in the database state.

first-assoc-role: $(G\text{-schema} \rightarrow (A \times R))$
next-assoc-role: $((G\text{-schema} \times A \times R) \rightarrow (A \times R))$
last-assoc-role: $(G\text{-schema} \rightarrow (A \times R))$

These three functions specify the ordering of role set edges in the database schema.

first-aid-role: $(G\text{-state} \rightarrow (A\text{-id} \times R))$
next-aid-role: $((G\text{-state} \times A\text{-id} \times R) \rightarrow (A\text{-id} \times R))$
last-aid-role: $(G\text{-state} \rightarrow (A\text{-id} \times R))$

These three functions specify the ordering of role edges in the database state.

For the three groups of functions above, we assume there is a "dummy" last element in each ordering which does not represent an actual edge.

image: $[[x \rightarrow y] \rightarrow \text{Powerset}(y)]$
dom: $[[x \rightarrow y] \rightarrow \text{Powerset}(x)]$

These functions return the domain and image of a function.

$X: ((x \times [y \rightarrow z]) \rightarrow [(x \times y) \rightarrow z])$

This function combines a constant into the domain of a function.

$||: (((x \rightarrow y) \times \text{Powerset}(x)) \rightarrow [x \rightarrow y])$

This function returns a function which is the same as its first argument restricted to the domain specified by the second argument.

head: $x^* \rightarrow x$
tail: $x^* \rightarrow x^*$

These functions return the first element and the remainder of a list respectively.

Figure A1.2 - Built-in functions

valid-entity(*schema*:G-schema, *state*:G-state, *e-id*:E-id, *e-chars*:Char-function)
returns Boolean

% check validity of entity description %

```
return    ¬(e-id ∈ state.E)                                % not already entity with same id %
          ∧ e-id.type ∈ schema.E                            % valid type %
          ∧ e-id.chars = e-chars || dom(e-id.chars)          % id matches other statement of characteristics in description %
          ∧ dom(e-chars) = {c:C | (e-id.type,c) ∈ dom(schema.c)} % correct characteristics defined %
          ∧ dom(e-id.chars) = {c:C | (e-id.type,c) ∈ dom(schema.c) % proper id %
          ∧ schema.fc(e-id.type,c) = key}
          ∧ ¬(∃c:C) (c ∈ dom(e-chars)                    % characteristic values in correct domain %
          ∧ ¬[e-chars(c).type = schema.c(e-id.type,c)
          ∧ schema.sd(e-chars(c).type)(e-chars(c).value) = true] )
```

Figure A1.3 - valid-entity Predicate

valid-association(schema:G-schema, state:G-state, a-id:A-id, a-roles:Role-function,
a-chars:Char-function)

returns Boolean

% check validity of association description %

```

return  ¬(a-id ∈ state.A)                % not already association with same id %
        ∧ a-id.type ∈ schema.A                % valid type %
        ∧ a-id.roles = a-roles || dom(a-id.roles)
                                                % id roles match those in description %
        ∧ a-id.chars = a-chars || dom(a-id.chars)
                                                % characteristics match those in description %
        ∧ dom(a-chars) = {c:C | (a-id.type,c) ∈ dom(schema.c)}
                                                % correct characteristics defined %
        ∧ dom(a-id.chars) = {c:C | (a-id.type,c) ∈ dom(schema.c)
                                ∧ schema.fc(a-id.type,c) = key}
                                                % correct characteristics in id %
        ∧ dom(a-id.roles) = {r:R | (a-id.type,r) ∈ dom(schema.r)
                                ∧ schema.fr(a-id.type,r).restrict = restricted}
                                                % correct roles in id %
        ∧ ¬(∃c:C) (c ∈ dom(a-chars)
                    % characteristic values in correct domain %
                    ∧ ¬[a-chars(c).type = schema.c(a-id.type,c)
                        ∧ schema.sd(a-chars(c).type)(a-chars(c).value)=true])
        ∧ ¬(∃r:R) (r ∈ dom(a-roles)
                    % characteristics of roles in correct domains %
                    ∧ ¬[a-roles(r).type = schema.r(a-id.type,r)
                        ∧ dom(a-roles(r).chars) = {c:C | (a-roles(r).type,c) ∈ dom(schema.c)
                                                            ∧ schema.fc(a-roles(r).type,c) = key}
                        ∧ ¬(∃c:C) (c ∈ dom(a-roles(r).chars)
                                    ∧ ¬[a-roles(r).chars(c).type = schema.c(a-id.type,c)
                                        ∧ schema.sd(a-roles(r).chars(c).type)(a-roles(r).chars(c).value)=true])])])

```

Figure A1.4 - valid-association Predicate

```

Insert-ind-entity (schema:G-schema, state:G-state, e-desc:E-desc)
  returns G-state

% insert independent entity %

let e-id = e-desc.E-id
    e-chars = e-desc.chars

in if valid-entity (schema, state, e-id, e-chars)           % proper entity description %
     $\wedge \neg (\exists r:R) (\exists a:A) (schema.r(a,r) = e-id.type$            % check that independent %
     $\wedge schema.fr(a,r).mand = mandatory)$ 

    then return (state.E U {e-id},                           % insert entity id %
                 state.A,
                 state.D U image(e-chars),                   % insert new domain nodes %
                 state.r,
                 state.c U X(e-id,e-chars) )                 % insert characteristic edges %

    else return error

```

Figure A1.5 - Insert-ind-entity Operation Type


```
Insert-ind-association (schema:G-schema, state:G-state, a-desc:A-desc)
  returns G-state

% insert independent association %

let a-id = a-desc.A-id
    a-roles = a-desc.roles
    a-chars = a-desc.chars

in if valid-association (schema, state, a-id, a-roles, a-chars)
    % proper association description %
     $\wedge \text{image}(a\text{-id.roles}) \subseteq \text{state.E}$  % all entities in association already in state %
then return (state.E,
    state.A  $\cup$  {a-id}, % insert association id %
    state.D  $\cup$  image(a-chars), % insert new domain nodes %
    state.r  $\cup$  X(a-id,a-roles), % insert role edges %
    state.c  $\cup$  X(a-id,a-chars) ) % insert characteristic edges %

else return error
```

Figure A1.6 - Insert-ind-association Operation Type

Insert-semantic-unit (schema:G-schema, state:G-state, arg-list:Arg-list)
returns G-state

% this initiates mutually recursive calls of insert-mand-entity, %

% insert-mand-association and check-roles in order to delete a semantic unit %

let (state',arg-list') = insert-mand-entity (schema,state,arg-list)

% insert first entity in argument list (which must be mandatory) %

in if arg-list' = nil

% all arguments properly inserted %

then return state'

else return error

Figure A1.7 - Insert-semantic-unit Operation Type (main procedure)

```

insert-mand-entity (schema:G-schema, state:G-state, arg-list:Arg-list)
  returns (G-state,Arg-list)

% insert dependent entity %

let e-id = head(arg-list).E-id           % get description from argument list %
    e-chars = head(arg-list).chars

in if valid-entity (schema, state, e-id, e-chars)           % valid entity description %
    ^ (∃r:R) (∃a:A) (schema.r(a,r) = e-id.type           % not independent %
    ^ schema.fr(a,r).mand = mandatory)

then let state' = ( state.E U {e-id},           % insert entity id %
    state.A,
    state.D U image(e-chars),   % insert new domain nodes %
    state.r,
    state.c U X(e-id,e-chars) ) % insert characteristic edges %

in return insert-mand-association(schema,state',tail(arg-list),
    e-id,first-assoc-role(schema))
    % next element on list should be mandatory association for this entity. %
    % to find which role and association this should be, start checking %
    % with "first" association, role pair in schema %

else return error

```

Figure A1.8 - insert-mand-entity Function


```

insert-mand-association (schema:G-schema, state:G-state, arg-list:Arg-list,
                        e-id:E-id, (a:A,role:R) )

    returns G-state

% for a given entity, finds and inserts all roles in associations which are %
% mandatory for this entity %

if (a,role) = last-assoc-role(schema)
    % if last, then can be no more mandatory roles for this entity %
    then return (state,arg-list)
else if schema.fr(a,r).mand = mandatory % check if role is mandatory %
    ^ -(∃a-id:A-id) (state.r(a-id,r) = e-id
    ^ a-id.type = a)

    then let a-id = head(arg-list).A-id
        a-roles = head(arg-list).roles
        a-chars = head(arg-list).chars
        % if mandatory, association should be next on list %

    in if valid-association (schema, state, a-id, a-roles, a-chars)
        % valid description %

        ^ a-id.type = a % proper type %
        ^ a-roles(role) = e-id % entity is in this association %

        then let state' = (state.E,
            state.A U {a-id}, % insert association %
            state.D U image(a-chars), % domain nodes %
            state.r U X(a-id,a-roles) % role edges %
            state.c U X(a-id,a-chars) )
            % insert characteristic edges %

        in let (state'',arg-list') =
            check-roles(schema,state',a-id,tail(arg-list),
                first-role(a,schema))
            % check each role in association to see if entity has been inserted %

            in return insert-mand-association(schema,state'',arg-list',
                e-id,next-assoc-role(schema,a,role) )
            % check next association, role pair to see if mandatory for this entity %

        else return error

    else return insert-mand-association(schema,state,arg-list,
        e-id,next-assoc-role(schema,a,role) )
        % role was not mandatory, check next association, role pair %

```

Figure A1.9 - insert-mand-association Function

```

check-roles (schema:G-schema, state:G-state, a-id:A-id,arg-list:Arg-list, role:R)
  returns (G-state,Arg-list)

% for a given association, checks each role to see if entity for role %
% still needs to be inserted (e.g. there are other dependent entities %
% in this association). %

if role = last-role(a-id.type,schema)                                % no more roles %
  then return (state,arg-list)

  else let e-id = state.r(a-id,role)

    in if e-id < state.E                                              % entity already inserted ? %

      then return check-roles(schema,state,a-id,arg-list,
                               next-role(a-id.type,schema,role))
                                                                    % yes, check next role %

      else let (state',arg-list') = insert-mand-entity(schema,state,arg-list)
                                                                    % no, insert the entity %

    in return check-roles(schema,state',a-id,arg-list',
                           next-role(a-id.type,schema,role))
                                                                    % check next role %

```

Figure A1.10 - check-roles Function

```

Delete-ind-entity (schema:G-schema, state:G-state, e-id:E-id)
  returns G-state
% delete independent entity %

if e-id ∈ state.E                                     % exists in state %
  ∧ ¬(∃r:R) (∃a-id:A-id) (state.r(a-id,r) = e-id)    % not in any associations %
  then return (state.E - {e-id},                      % delete entity id %
              state.A,
              state.D - {d:D | (∃c:C) (state.c(e-id,c)=d
                                   ∧ ¬(∃x:Id) (x ≠ e-id
                                   ∧ state.c(x,c) = d) ) },
              state.r,
              state.c || {(x:Id,c:C) | (x,c) ∈ dom(state.c) ∧ x ≠ e-id} )
                                   % delete domain nodes not used elsewhere %
                                   % delete characteristic edges %
  else return error

```

Figure A1.11 - Delete-ind-entity Operation Type


```

Delete-ind-association (schema:G-schema, state:G-state, a-id:A-id)
  returns G-state

% delete independent association %

if a-id ∈ state.A
  % exists in state %
  % independent %
  ∧ ¬(∃r:R) ((a-id,r) ∈ dom(state.r)
    ∧ schema.fr(a-id.type,r).mand = mandatory
    ∧ ¬(∃a-id':A-id) (state.r(a-id',r) = state.r(a-id,r)
      ∧ a-id'.type = a-id.type
      ∧ a-id' ≠ a-id))

  then return (state.E,
    state.A - {a-id},
    state.D - {d:D | (∃c:C) (state.c(a-id,c)=d
      ∧ ¬(∃x:Id) (x ≠ a-id
        ∧ state.c(x,c) = d) ) },
    % delete domain nodes not used elsewhere %
    state.r || {(a-id':A-id,r:R) | (a-id',r) ∈ dom(state.r)
      ∧ a-id ≠ a-id'},
    % delete role edges %
    state.c || {(x:Id,c:C) | (x,c) ∈ dom(state.c)
      ∧ x ≠ a-id},
    % delete characteristic edges %

  else return error

```

Figure A1.12 - Delete-ind-association Operation Type

```

Delete-semantic-unit (schema:G-schema, state:G-state, a-id:A-id)
  returns G-state

% delete semantic unit by mutually recursive calls of Delete-semantic-unit, %
% delete-roles and delete-mand-entity %

if a-id ∈ state.A % association in state %

  then if ¬(∃r:R) ((a-id,r) ∈ dom(state.r)
    ∧ schema.tr(a-id.type,r).mand = mandatory
    ∧ ¬(∃a-id':A-id) (state.r(a-id',r) = state.r(a-id,r)
      ∧ a-id'.type = a-id.type
      ∧ a-id' ≠ a-id))
    % error if association not in semantic unit %

    then return error

  else let state' = (state.E,
    state.A - {a-id}, % delete association id %
    state.D,
    state.r,
    state.c)

    in let state'' = delete-roles(schema,state',a-id,
      first-role(a-id.type,schema))
      % start checking roles to delete dependent entities in association %

      in return (state''.E,
        state''.A,
        state''.D - {d:D | (∃c:C) (state''.c(a-id,c)=d
          ∧ ¬(∃x:Id) (x ≠ a-id
            ∧ state''.c(x,c) = d) ) },
          % delete domain nodes not used elsewhere %
        state''.r,
        state''.c || {(x:Id,c:C) | (x,c) ∈ dom(state''.c)
          ∧ x ≠ a-id} )
          % delete characteristic edges %

    else return error
  
```

Figure A1.13 - Delete-semantic-unit Operation Type (main procedure)

```

delete-roles (schema:G-schema, state:G-state, a-id:A-id, role:role)
  returns G-state

% delete each dependent role in association %
if role = last-role(a-id.type,schema)                                % no more roles %
  then return state                                                  % done %
else let e-id = state.r(a-id,role)
  state' = (state.E,
            state.A,
            state.D,
            state.r || {(a-id':A-id,r:R) | (a-id',r) ∈ dom(state.r)
                        ∧ (a-id',r) ≠ (a-id,role)}),
            % delete role edge %
            state.c)

  in if e-id ∈ state'.E                                              % entity still in state %
    ∧ schema.fr(a-id.type,role).mand = mandatory
    ∧ ¬(∃a-id':A-id) (state.r(a-id',role) = e-id
                     ∧ a-id'.type = a-id.type
                     ∧ a-id' ≠ a-id)                                % dependent %
  then let state'' = (state'.E - {e-id},                             % delete entity id %
                     state'.A,
                     state'.D - {d:D | (∃c:C) (state'.c(e-id,c)=d
                                                ∧ ¬(∃x:Id) (x = e-id
                                                             ∧ state'.c(x,c) = d))}),
                     % delete domain nodes not used elsewhere %
                     state'.r,
                     state'.c || {(x:Id,c:C) | (x,c) ∈ dom(state'.c)
                                   ∧ x = e-id} )
                     % delete characteristic edges %
  in return delete-roles(schema,                                     % check next role %
                        delete-mand-entity(schema,
                                             state'',
                                             e-id,
                                             first-aid-role(state''))
                        % delete dependent entity %
                        a-id,
                        next-role(a-id.type,schema,role))

else return delete-roles(schema,state',a-id,                       % check next role %
                        next-role(a-id.type,schema,role))

```

Figure A1.14 - delete-roles Function


```

delete-mand-entity (schema:G-schema, state:G-state, e-id:E-id, (a-id:A-id,role:R))
  returns G-state

% checks to see if entity is mandatory in other semantic units %

if (a-id,role) = last-aid-role(state)                                     % last role %

  then return state                                                         % done %

else if state.r(a-id,role) = e-id                                         % entity in association %
  ^ a-id ∈ state.A                                                         % association in state %

  then if schema.fr(a-id.type,role).mand = mandatory                     % dependent %
    ^ ¬(∃a-id':A-id) (state.r(a-id',role) = state.r(a-id,role)
      ^ a-id'.type = a-id.type
      ^ a-id' ≠ a-id)

    then return delete-mand-entity(schema,
      Delete-semantic-unit(schema,state,a-id),
      % delete the semantic unit containing this entity and association %
      e-id,
      next-aid-role(state,a-id,role))
      % check next role %

    else return error

  else return delete-mand-entity(schema,                                  % check next role %
    state,
    e-id,
    next-aid-role(state,a-id,role))

else return error

```

Figure A1.15 - delete-mand-entity Function

Appendix 2. The Semantic Relation Data Model: Formal Operation Specifications

R-schema	= (Rel: [R-names \rightarrow Heading], Con: Powerset(Constraints), R-names, P, S, E, D, C, sd: [D \rightarrow [Str \rightarrow Boolean]], Str)
R-state	= [R-names \rightarrow Relation]
Heading	= [H \rightarrow [C \rightarrow D]]
H	= (Powerset' (P \times S) \times E) \cup P
Relation	= Powerset(Tuple)
Tuple	= [H \rightarrow Charfunction]
Charfunction	= [C \rightarrow Domain-element]
Domain-element	= (type:D, value:Str)
Case-col	= (predicate-case:Powerset' (P \times S), type:E)
Col-list	= Case-col*
Constraint	= [(R-names \rightarrow Heading), R-State] \rightarrow Boolean]
Boolean	= {true, false}
R-names	= relation names
P	= predicates
S	= semantic case names
E	= entity set names
D	= domain names
C	= characteristics
Str	= character strings

Figure A2.1 - Type Specifications

image: $[[x \rightarrow y] \rightarrow \text{PowerSet}(y)]$

dom: $[[x \rightarrow y] \rightarrow \text{PowerSet}(x)]$

These functions return the domain and image of a function.

X: $[(x \times [y \rightarrow z]) \rightarrow [(x \times y) \rightarrow z]]$

This function combines a constant into the domain of a function.

||: $[[([x \rightarrow y] \times \text{PowerSet}(x)) \rightarrow [x \rightarrow y]]$

This function returns a function which is the same as its first argument restricted to the domain specified by the second argument.

head: $x^* \rightarrow x$

tail: $x^* \rightarrow x^*$

These functions return the first element and the remainder of a list respectively.

<x | P(x)>

This means "the unique x such that $P(x)$ ". Evaluates to error if there is no such x or there is more than one such x .

$(x_1, x_2, \dots, x_n) \downarrow i = x_i$ (selection function)

Figure A2.2 - Built-in functions and notation

Definition A2.1: Ordering of Domain Elements

For domain element types $d, d' \in D$ and strings $x, y \in \text{Str}$,
 $(d, x) \geq (d, y)$ if and only if

1. $d = d'$
- and 2.
 - i). $x = y$
 - or ii). $x = \text{"---"} \text{ and } y = \text{"---"}.$

Definition A2.2: Ordering of Column-group Headings (Elements of H)

For $h, h' \in H$, $p, p' \in P$, $\text{pc-set}, \text{pc-set}' \in \text{Powerset}'(P \times S)$ and $e \in E$,
 $h \geq h'$ if and only if

1. $h = h'$
- or 2.
 - i. $h = (\text{pc-set}, e)$ and $h' = (\text{pc-set}', e)$
 - and ii. $\text{pc-set} \geq \text{pc-set}'$

Definition A2.3: Ordering of Tuples

For $t, t' \in \text{Tuple}$,

1. If $\text{dom}(t) = \text{dom}(t')$,
then $t \geq t'$ if and only if

$$(\forall h: H) (h \in \text{dom}(t) \supset \text{dom}(t(h)) = \text{dom}(t'(h))) \\
\wedge (\forall c: C) (c \in \text{dom}(t(h)) \supset t(h)(c) \geq t'(h)(c))$$

This is for the case of tuples with identical headings.

2. If $\text{dom}(t) \neq \text{dom}(t')$,
then $t \geq t'$ if and only if
there is a 1-1 onto correspondence $f: \text{dom}(t) \rightarrow \text{dom}(t')$ such that

$$(\forall h: H) (h \in \text{dom}(t) \supset \text{dom}(t(h)) = \text{dom}(t'(f(h)))) \\
\wedge (\forall c: C) (c \in \text{dom}(t(h)) \supset t(h)(c) \geq t'(f(h))(c))$$

This is for the case of non-identical headings.

Figure A2.3 - Ordering Definitions

AD-A066 386

MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 5/2
EQUIVALENCE PROPERTIES OF SEMANTIC DATA MODELS FOR DATABASE SYS--ETC(U)
JAN 79 S A BORKIN N00014-75-C-0661
MIT/LCS/TR-206 NL

UNCLASSIFIED

4 OF 4
ADA
066386

11 11 11



END

DATE
FILMED

5-79
DDC

```
valid-rels (schema:R-schema, rels:R-state)
  returns Boolean

% checks that relations are of proper form %

return dom(rels) = dom(schema.Rel)
  ∧ [(∀r-n:R-names) (r-n ∈ dom(rels)
    ▷ [(∀t:Tuples) (t ∈ rels(r-n)          % for each tuple in each relation %
      ▷ [dom(t) = dom(schema.Rel(r-n))      % correct column-groups %
        ∧ (∃h:H) (∃c:C) (h ∈ dom(t)         % at least one non-null value %
          ∧ c ∈ dom(t(h)) ∧ t(h)(c).value ≠ "----")
        ∧ (∀h:H) (h ∈ dom(t)
          ▷ [dom(t(h)) = dom(schema.Rel(r-n)(h))
            % correct characteristics %
            ∧ (∀c:C) (c ∈ dom(t(h))
              ▷ [t(h)(c).type = schema.Rel(r-n)(h)(c)
                ∧ schema.sd(schema.Rel(r-n)(h)(c))(t(h)(c).value) = true]]))]]
            % values in proper domains %
          ]
        ]
      ]
    ]
  )
  ∧ ¬(∃t:Tuples) (∃t':Tuples) (t ∈ rels(r-n) ∧ t' ∈ rels(r-n) ∧ t > t'))
  % all tuples are incomparable under partial order %
```

Figure A2.4 - valid-rels Predicate.


```

Insert-tuples(schema:R-schema, state:R-state, addition:R-state)
  returns R-state

% insert statements into relations %

if valid-rels(schema,addition)                                % addition in proper form %
  ^ valid-rels(schema,state)                                    % state in proper form %
  ^ (∀r-n:R-names) (r-n ∈ dom(addition)
    > ¬(∃t:Tuple) (∃t':Tuple) (t ∈ addition(r-n) ∧ t' ∈ state(r-n)
      ^ t' ≥ t))
    % not attempting to insert statements already there %
    % or statements less than those already in state %
  then let state' = λr.(s-union(state(r), addition(r)))          % do insertion %
    in if (∀con:Constraint) (con ∈ schema.Con                    % check constraints %
      > con(schema.Rel,state') = true)
      then return state'                                          % ok %
      else return error                                           % constraints failed %
  else return error

```

Figure A2.5 - Insert-tuples Operation Type

```

s-union(r-1:Relation,r-2:Relation)
  returns Relation

% performs union of relations, deleting statements which %
% are less than some other statement %

return {t:Tuple | t ∈ r-1 ∪ r-2
  ^ ¬(∃t':Tuple) (t' ∈ r-1 ∪ r-2 ∧ t' > t)}

```

Figure A2.6 - s-union Function

```

Delete-tuples(schema:R-schema, state:R-state, deletion:R-state)
  returns R-state

% delete tuples from relations %

if valid-reis(schema,state)                                     % state in proper form %
  ^ (Vr-n:R-names) (r-n ∈ dom(addition)
    > (Vt:Tuple) (t ∈ deletion(r-n)
      > (∃t':Tuple) (t' ∈ state(r-n) ∧ t' ≥ t)))
    % each tuple in deletion is less than or equal to some tuple in state %
  then let state' = λr.(s-difference(state(r), deletion(r))) % perform deletion %
    in if (Vcon:Constraint) (con ∈ schema.Con                  % check constraints %
      > con(schema.Rel,state') = true)
      then return state'                                       % ok %
      else return error                                         % constraints failed %
    else return error

```

Figure A2.7 - Delete-tuples Operation Type

s-difference(r-1:Relation,r-2:Relation)
returns Relation

% delete statements from relations %

% make all statements in r-1 less than statements in r-2 %

% by changing values to null or deleting statement if necessary %

return (r-1 - {t:Tuple | (∃t':Tuple) (t ∈ r-1 ∧ t' ∈ r-2 ∧ t ≥ t')})

% remove tuples greater and equal to those in deletion %

U {t':Tuple | (∃t:Tuple) (∃t':Tuple) (t ∈ r-1 ∧ t' ∈ r-2 ∧ t > t')}

% statement in r-1 is greater than statement in r-2 %

% add back lesser statement (if one exists) to r-1 %

∧ (∀h:H) (∀h':H) (∀c:C)

(h ∈ dom(t) ∧ c ∈ dom(t(h)) ∧ h' ∈ dom(t') ∧ c ∈ dom(t'(h'))

⇒ ((h = h' ∧ t(h)(c) = t'(h)(c))

⇒ t''(h)(c) = "----")

% the case of equal headings and values - change value to null %

∧ ((h > h' ∨ (h = h' ∧ t(h)(c) > t'(h)(c)))

⇒ t''(h)(c) = t(h)(c)))

% otherwise leave value alone %

∧ dom(t'') = dom(t)

% correct form for resulting tuple %

∧ (∀h:H) (h ∈ dom(t) ⇒ dom(t''(h)) = dom(t(h)))

∧ (∃h:H) (∃c:C) (h ∈ dom(t'')

% at least one non-null value %

∧ c ∈ dom(t''(h)) ∧ t''(h)(c).value ≠ "----"))

Figure A2.8 - s-difference Function

Appendix 3. The Semantic Relation Data Model: Formal Constraint Specifications

```

Project(new-head:Heading, (head:Heading, r:Relation))
  returns (Heading, Relation)

% eliminates columns and/or predicate:case pairs from heading %

if dom(new-head) = ∅                                     % cannot project to empty heading %
  ∧ (∀h:H) (h ∈ dom(new-head))                          % check form of new heading %
    > (∃h':H) (h' ∈ dom(head) ∧ h ≤ h'
      ∧ dom(h) ⊆ dom(h')
      ∧ (∀c:C) (c ∈ dom(h)
        > h(c) = h'(c))))

then let r' = {t:Tuples | dom(t) = dom(new-head)         % tuples of proper form %
  ∧ (∀h:H) (h ∈ dom(t)
    > dom(t(h)) = dom(new-head(h))

  ∧ (∃h:H) (∃c:C) (h ∈ dom(t) % exists at least one non-null value %
    ∧ c ∈ dom(t(h)) ∧ t(h)(c) ≠ """)

  % values in projected tuple equal to those in tuple of r %
  ∧ (∃t':Tuples) (t' ∈ r
    ∧ (∀h:H) (∀c:C)
      ((h ∈ dom(t) ∧ c ∈ dom(t(h)))
        > (t(h)(c) = t'(<h':H|h' ∈ dom(head) ∧ h ≤ h'>)(c))))))

in return (new-head, r')

else return error

```

Figure A3.1 - Project Constraint Operation Type

```
Equal(heading-1:Heading,r-1:Relation),(heading-2:Heading,r-2:Relation))
returns Boolean
```

% checks if two relations are identical %

```
if heading-1=heading-2  $\wedge$  r-1 = r-2
```

```
then return true
```

```
else return false
```

Figure A3.2 - Equal Constraint Type

```
Col-func(heading-1:Heading,heading-2:Heading,(heading:Heading,rel:Relation))
returns Boolean
```

% Checks that no two tuples in *rel* are equal in columns specified %

% by *heading-1*, but unequal in columns specified by *heading-2*. %

% It is not required that tuples with null values in the columns of *heading-1* %

% need have equal values in *heading-2* %

```
if  $\neg(\exists t)(\exists t')(t \in rel \wedge t' \in rel$ 
```

```
 $\wedge (\forall h:H)(\forall c:C)(h \in dom(heading-1) \wedge c \in dom(heading-1(h))$ 
```

```
 $\supset t(h)(c) = t'(h)(c) \wedge t(h)(c) \neq \text{---})$ 
```

```
 $\wedge (\exists h:H)(\exists c:C)(h \in dom(heading-2) \wedge c \in dom(heading-2(h))$ 
```

```
 $\wedge t(h)(c) \neq t'(h)(c))$ 
```

```
then return true
```

```
else return false
```

Figure A3.3 - Col-func Constraint Type

```
Subset((heading-1:Heading,r-1:Relation),(heading-2:Heading,r-2:Relation))  
  returns Boolean
```

```
% Check that characteristic values in r-1 tuples are a subset %  
% of those in r-2 tuples. %  
% Does not require that characteristics describe same predicate:case pair %  
% or same predicate. %
```

```
if |dom(heading-1)|=|dom(heading-2)|-1  
  % each relation may describe only a single predicate or %  
  % set of predicate:case pairs %
```

```
then if {cf | (∃t:tuple)(tcr-1 ∧ {cf}=image(t))}  
  ⊆ {cf | (∃t:tuple)(tcr-2 ∧ {cf}=image(t))}
```

```
  then return true
```

```
  else return false
```

```
else return error
```

Figure A3.4 - Subset Constraint Type


```
Equalc((heading-1:Heading,r-1:Relation),(heading-2:Heading,r-2:Relation))  
  returns Boolean
```

```
% check that characteristic values in r-1 tuples are equal %  
% to those in r-2 tuples. %  
% Does not require that characteristics describe same predicate:case pair %  
% or same predicate. %
```

```
  if |dom(heading-1)|=|dom(heading-2)|-1  
    % each relation may describe only a single predicate or %  
    % set of predicate:case pairs %
```

```
  then if {cf | (∃t:tuple) (t ∈ r-1 ∧ {cf} = image(t))}  
    = {cf | (∃t:tuple) (t ∈ r-2 ∧ {cf} = image(t))}
```

```
    then return true
```

```
    else return false
```

```
  else return error
```

Figure A3.5 - Equalc Constraint Type

Non-null(*heading-1*:Heading,*r-1*:Relation)
returns Boolean

% checks that no null values are present in relation %

if $\neg(\exists t:\text{Tuple}) (\exists h:H) (\exists c:C) (t \in r-1$
$\wedge h \in \text{dom}(t)$
$\wedge c \in \text{dom}(t(h)) \wedge t(h)(c) = \text{"---"})$

then return true

else return false

Figure A3.6 - Non-null Constraint Type

```

Case-join (col-2:Case-col,col-1:Case-col,(heading-2:Heading,r-2:Relation),
          (heading-1:Heading,r-1:Relation))
  returns (Heading,Relation)

% Add to col-1 in r-1 those characteristics defined in col-2 of r-2 %
% Match tuples according to equal values in common characteristics %
% of col-1 and col-2 %

if col-1 ∈ dom(heading-1)
  ∧ col-2 ∈ dom(heading-2)
  ∧ col-1.type = col-2.type                                     % same entity type %
  ∧ (∀c:⊂) (c ∈ dom(heading-1(col-1)) ∧ c ∈ dom(heading-2(col-2))
    ⇒ heading-1(col-1)(c) = heading-2(col-2)(c))
    % same domains for common characteristics %

  then return ( (λh'. if h' = col-1                               % heading definition %
    then return (λc'.if c' ∈ dom(col-1)
      then return col-1(c')
      else return col-2(c'))
    else return heading-1(h')),

    (t:Tuple | (∃t':Tuple) (∃t'':Tuple)                          % tuple definition %
      % consider tuple from each relation %
      (t' ∈ r-1 ∧ t'' ∈ r-2
        ∧ dom(t) = dom(t')
        ∧ dom(t(col-1)) = dom(t'(col-1)) ∪ dom(t''(col-2))
        % add characteristics from r-2 %
        ∧ (∀h:H) (h=col-1 ⇒ t(h)=t'(h))
        % values equal to those in r-1 for all columns other than those joined %
        ∧ (∀c:⊂) ((c ∈ dom(t'(col-1))
          ⇒ t'(col-1)(c)=t(col-1)(c))
          % values equal to those in r-1 for characteristics defined in r-1 %
          ∧ (c ∈ dom(t''(col-2)) ⇒ t''(col-2)(c)=t(col-1)(c))))))
        % values equal to those in r-2 for characteristics defined in r-2 %

    else return error

```

Figure A3.7 - Case-join Constraint Operation Type


```

Predicate-join(pred:P, (heading-2:Heading, r-2:Relation),
               (heading-1:Heading, r-1:Relation))
  returns (Heading, Relation)

% Add to r-1 the information in r-2 concerning the predicate pred %
% Match tuples according to equal values for common cases and characteristics %
% of pred %

if ( $\forall c:C$ ) ( $c \in \text{dom}(\text{heading-1}(\text{pred})) \wedge c \in \text{dom}(\text{heading-2}(\text{pred}))$ 
            $\supset \text{heading-1}(\text{pred})(c) = \text{heading-2}(\text{pred})(c)$ )
  % predicate characteristics common to both relations defined on same domains %
  % all cases of pred common to both relations are defined for same %
  % entity type, same characteristics and same domains %
   $\wedge (\forall \text{col}:\text{Case-col}) (\forall \text{col}':\text{Case-col}) (\forall s:S)$ 
    ( $\text{col} \in \text{dom}(\text{heading-1}) \wedge (\text{col}' \in \text{dom}(\text{heading-2}))$ 
      $\wedge (\text{pred}, s) \in \text{col}.\text{predicate-case}$ 
      $\wedge (\text{pred}, s) \in \text{col}'.\text{predicate-case}$ 
      $\supset \text{col.type} = \text{col}'.\text{type}$ 
      $\wedge \text{heading-1}(\text{col}) = \text{heading-2}(\text{col}')$ )

  % pred not combined with any other predicate in any column of r-2 %
   $\wedge (\forall \text{col}':\text{Case-col}) (\forall s:S)$ 
    ( $\text{col}' \in \text{dom}(\text{heading-2}) \wedge (\text{pred}, s) \in \text{col}'.\text{predicate-case}$ 
      $\supset \text{col}'.\text{predicate-case} = \{(\text{pred}, s)\}$ )

  then let new-cols = {col:Case-col | ( $\exists s:S$ ) (col  $\in \text{dom}(\text{heading-2})$ 
     $\wedge \{(\text{pred}, s)\} = \text{col}.\text{predicate-case}$ 
     $\wedge \neg (\exists \text{col}':\text{Case-col}) (\text{col}' \in \text{dom}(\text{heading-1})$ 
     $\wedge (\text{pred}, s) \in \text{col}'.\text{predicate-case})$ )}
    % note those cases of pred in r-2 which are not present in r-1 %

  in let new-head =  $\lambda h'. \text{if } h' = \text{pred}$  % define new heading %
    then return ( $\lambda c'. \text{if } c' \in \text{dom}(\text{col-1})$ 
      then return col-1(c')
      % pred characteristics defined in r-1 %
      else return col-2(c'))
    % pred characteristics defined in r-2 but not r-1 %
    else if h'  $\in \text{new-cols}$ 
      then return heading-2(h') % new cases %
      else return heading-1(h')

```

Figure A3.8 - Predicate-join Constraint Operation Type
(Continued on Next Page)

```

% define tuples of resulting relation %
in let r' = {t:Tuple | (∃t':Tuple) (∃t'':Tuple)
% consider tuple from each relation %
(t' ∈ r-1 ∧ t'' ∈ r-2
∧ dom(t) = dom(t') ∪ new-cols ∪ {p:P | p=pred ∧ p ∈ dom(t'')}
∧ dom(t(pred)) = dom(t'(pred)) ∪ dom(t''(pred))
∧ (∀h:H) ((h ∈ dom(t')-pred ⇒ t'(h) = t(h))
% values equal to those in r-1 %
∧ (h ∈ new-cols ⇒ t(h) = t''(h))
% values of cases for pred not in r-1 equal to those in r-2 %
% values equal to those in r-2 for cases of pred %
∧ (∃col:Case-col) (col ∈ dom(t'')
∧ h ∈ dom(t) ∧ col ≤ h
⇒ t''(col) = t(h))
% values equal to those in r-1 and r-2 for characteristics of pred %
∧ (∀c:C)
((c ∈ dom(t'(pred))
⇒ t'(pred)(c) = t(pred)(c))
∧ (c ∈ dom(t''(pred))
⇒ t''(pred)(c) = t(pred)(c)))));
in return (new-head, r')
else return error

```

Figure A3.8 - Predicate-join Constraint Operation Type

```

Conjunction(col-list-1:Col-list,col-list-2:Col-list,
            (heading-1:Heading,r-1:Relation),
            (heading-2:Heading,r-2:Relation))
    returns (Heading,Relation)

% combine relations describing different predicates into a single relation. %
% The columns of col-list-1 in r-1 are combined in sequence with %
% the columns of col-list-2 in r-2 %

let n = length(col-list-1)
    m = length(col-list-2)
    col-set-1 =  $\bigcup_{i=1}^n \{col-list-1\}i$ 
    col-set-2 =  $\bigcup_{i=1}^m \{col-list-2\}i$ 

in if col-set-1  $\subseteq$  dom(heading-1)  $\wedge$  col-set-2  $\subseteq$  dom(heading-2)
     $\wedge$  n = m  $\wedge$  |col-set-1| = n  $\wedge$  |col-set-2| = m
        % lists of same length with no duplicates %
         $\wedge$  ( $\forall 1 \leq i \leq n$ ) ((col-list-1i).type = (col-list-2i).type
             $\wedge$  heading-1(col-list-1i) = heading-2(col-list-2i))
            % corresponding columns describe same entity type and same characteristics %
        % check that sets of predicates in the two relations are mutually exclusive %
         $\wedge$   $\neg(\exists p:P)(\exists s:S)(\exists s':S)(\exists col:Case-col)(\exists col':Case-col)$ 
            ((p,s)  $\in$  col.predicate-case
             $\wedge$  (p,s')  $\in$  col'.predicate-case
             $\wedge$  col  $\in$  dom(heading-1)
             $\wedge$  col'  $\in$  dom(heading-2))
         $\wedge$   $\neg(\exists p:P)(p \in$  dom(heading-1)
             $\wedge$  p  $\in$  dom(heading-2))

    then let col-list-new = cons $_{i=1}^n$  ((col-list-1i).predicate-case U
        (col-list-2i).predicate-case, (col-list-1i).type)
        % new columns describe union of predicate:case pairs in joined columns %

    in let col-set-new =  $\bigcup_{i=1}^n \{col-list-new\}i$ 

        in let new-r = { t:Tuple | % define tuples in resulting relation %
            dom(t) = dom(heading-1) - col-set-1
                U dom(heading-2) - col-set-2
                U col-set-new

```

Figure A3.9 - Conjunction Constraint Operation Type
(Page 1 of 3)


```

% case 1 - match of equal values in joined columns %
^ (( $\exists t'$ :Tuple) ( $t' \in r-1$ 
  ^ ( $\forall h:H$ ) ( $(h \in \text{dom}(\text{heading-1}) - \text{col-set-1}$ 
     $\supset t'(h) = t(h)$ )
    ^ ( $h \in \text{col-set-1}$ 
       $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )
^ ( $\exists t'$ :Tuple) ( $t' \in r-2$ 
  ^ ( $\forall h:H$ ) ( $(h \in \text{dom}(\text{heading-2}) - \text{col-set-2}$ 
     $\supset t'(h) = t(h)$ )
    ^ ( $h \in \text{col-set-2}$ 
       $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )

% case 2 - no match in r-2 for tuple in r-1 - pad with null values %
v (( $\exists t'$ :Tuple) ( $t' \in r-1$ 
  ^ ( $\forall h:H$ ) ( $(h \in \text{dom}(\text{heading-1}) - \text{col-set-1}$ 
     $\supset t'(h) = t(h)$ )
    ^ ( $h \in \text{col-set-1}$ 
       $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )
^  $\neg(\exists t'$ :Tuple) ( $t' \in r-2$ 
  ^ ( $\forall h:H$ ) ( $h \in \text{col-set-2}$ 
     $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )
^ ( $\forall h:H$ ) ( $h \in \text{dom}(\text{heading-2}) - \text{col-set-2}$ 
   $\supset (\text{dom}(t(h)) = \text{dom}(\text{heading-2}(h))$ 
    ^ ( $\forall c:C$ ) ( $c \in \text{dom}(t(h)) \supset t(h)(c) = \text{"---"}))$ ))

% case 3 - no match in r-1 for tuple in r-2 - pad with null values %
v (( $\exists t'$ :Tuple) ( $t' \in r-2$ 
  ^ ( $\forall h:H$ ) ( $(h \in \text{dom}(\text{heading-2}) - \text{col-set-2}$ 
     $\supset t'(h) = t(h)$ )
    ^ ( $h \in \text{col-set-2}$ 
       $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )
^  $\neg(\exists t'$ :Tuple) ( $t' \in r-1$ 
  ^ ( $\forall h:H$ ) ( $h \in \text{col-set-1}$ 
     $\supset t'(h) = t(\langle h' \mid h' \in \text{col-set-new} \wedge h \leq h' \rangle))))$ )
^ ( $\forall h:H$ ) ( $h \in \text{dom}(\text{heading-1}) - \text{col-set-1}$ 
   $\supset (\text{dom}(t(h)) = \text{dom}(\text{heading-1}(h))$ 
    ^ ( $\forall c:C$ ) ( $c \in \text{dom}(t(h)) \supset t(h)(c) = \text{"---"}))$ ))

```

Figure A3.9 - Conjunction Constraint Operation Type
(Page 2 of 3)

```
in let new-head =  $\lambda h'. \text{if } h' \in \text{col-set-new} \quad \% \text{ define new heading } \%$   
    then return heading-1( $\langle h \# h' \mid h \in \text{col-set-1} \wedge h \leq h' \rangle$ )  
    else if  $h' \in \text{col-set-1}$   
        then return heading-1( $h'$ )  
        else return heading-2( $h'$ )  
  
in return (new-head, new-r)  
  
else return error
```

Figure A3.9 - Conjunction Constraint Operation Type
(Page 3 of 3)

Biographic Note

Sheldon Asher Borkin was born November 16, 1952 in Detroit, Michigan and was raised in Huntington Woods, Michigan. Mr. Borkin attended Berkley High School in Berkley, Michigan where he received a National Merit Award, the Science Department Award, and the Rensselaer Mathematics and Science Award. He was a member of the National Honor Society and president of the Science Research Project. He graduated high school *summa cum laude* in June 1970.

Beginning in September 1970, Mr. Borkin attended the University of Michigan in Ann Arbor where he was in the Honors Program. In May 1974, he received the Bachelor of Science degree with high distinction and departmental honors in the Computer and Communication Sciences Department. At the University of Michigan, Mr. Borkin received the Branstrom Prize, was elected a member of Phi Beta Kappa, and was named both a Regents-Alumni Scholar and an Angell Scholar.

Mr. Borkin has attended the Massachusetts Institute of Technology in Cambridge, Massachusetts since February 1975 where he has been associated with the Computation Structures Group of the Laboratory for Computer Science. He received the Master of Science and Electrical Engineer degrees in June 1978 from the Department of Electrical Engineering and Computer Science. His thesis title was "Data Model Equivalence." Mr. Borkin was a Teaching Assistant in the Department of Electrical Engineering and Computer Science from September 1976 until May 1978. In May 1978, he received a Departmental Teaching Award.

He was initiated into Sigma Xi, the honorary scientific research society in April 1976. Mr. Borkin is also a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers. He has been married to his wife, Sandra, since July 1973.

Mr. Borkin was employed at the Great Lakes Environmental Research Laboratory in Detroit, Michigan during the summers of 1972, 1973 and 1974. He was employed by Bell Telephone Laboratories in Piscataway, New Jersey from August 1974 until February 1975. During the summer of 1977, Mr. Borkin was associated with the Charles Stark Draper Laboratory in Cambridge, Massachusetts. Since December 1978, Mr. Borkin has been employed by the IBM Cambridge Scientific Center in Cambridge, Massachusetts where he is continuing research in the field of database systems and their user interfaces.

Mr. Borkin has published:

Borkin, S.A., "Data Model Equivalence", Proceedings of the Fourth International Conference on Very Large Data Bases, IEEE, September 1978. (also published as MIT Laboratory for Computer Science Technical Memo 118 and Computation Structures Group Memo 158)

OFFICIAL DISTRIBUTION LIST

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Electronics Lab Center
Advanced Software Technology
Division - Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Captain Richard L. Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York, N. Y. 09501
1 copy